DYNAMIC AND ELASTIC MEMORY MANAGEMENT IN VIRTUALIZED CLOUDS

A Dissertation
Presented to
The Academic Faculty

By

Wenqi Cao

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2019

DYNAMIC AND ELASTIC MEMORY MANAGEMENT IN VIRTUALIZED CLOUDS

Approved by:

Dr. Ling Liu, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. David Devecsery
School of Computer Science
Georgia Institute of Technology

Dr. Calton Pu, Co-Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Joy Arulraj
School of Computer Science
Georgia Institute of Technology

Dr. Jay Lofstead
Scalable System Software Group
Sandia National Laboratories

Date Approved: March 26, 2019

To my parents and my wife

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Dr. Ling Liu, for her guidance in this research and the support throughout my graduate study. She always inspires me to be a better researcher in many ways. I was especially fortunate to work with her and benefit from her advises. In addition to academic research, she always shares her life lessons with me and always been available whenever I need help. Her principles and dedication will have a great influence on me.

I would also like to give special thanks to Dr. Calton Pu, Dr. David Devecsery, Dr. Joy Arulraj, and Dr. Jay Lofstead for their time to serve on my thesis committee and their valuable feedback on my thesis proposal and final dissertation. Their welcome suggestions and insightful comments have greatly contributed to my thesis. I have been fortunate to spend one summer at IBM Research Almaden and two summers at IBM Research T.J. Watson as research intern and work on cutting-edge research topics. I would like to express my sincere gratitude to my mentors and collaborators, including Dr. Nagapramod Mandagere, Dr. Abdullah Kayi, Dr. Alessandro Morari, Dr. Yoonho Park, Dr. Gong Su, Dr. Arun Iyengar, and Dr. Donna Dillenberger.

Thank you to the every members of DiSL research group at Georgia Tech. It was a great pleasure to work in such a dynamic research environment. I convey special thanks to Qi Zhang, Semih Sahin, Lei Yu, Emre Yigitoglu, Yang Zhou, Juhyun Bae and Yanzhao Wu for countless research discussions and friendship.

Finally, and most importantly, I would like to thank my wife Feifei Wang for her love and constant support. She has made my Ph.D. journey worthwhile. We look forward to starting a new chapter in our lives together. Thank you for being my best friend. I owe you everything. I thank my parents, Bailin Cao and Zhumei Sheng, for encouraging me to go confidently in the direction of my dreams, endless patience, and unconditional support. I thank Feifei's parents, Guanbin Wang and Yingjie Feng. Their continuous support and

encouragement empower me to pursue my dream. Finally, I am especially grateful to my grandparents, Zeyuan Cao and Shuxian Zhang. Thank you so much for being in my life. I miss you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The memory capacity of computers and edge devices continue to grow: the DRAM capacity for low end computers are at tens or hundreds of GBs and the modern high performance computing (HPC) platforms can support terabytes of RAM for Big data driven HPC and Machine Learning (ML) workloads. Although system virtualization improves resource consolidation, it does not tackle the increasing cost of address translation and the growing size of page tables OS kernel maintains. All virtual machines and processors use pages tables for address translation. On the other hand, Big data and latency-demanding applications are typically deployed in virtualized Clouds using the application deployment models, comprised of virtual machines (VMs), containers, and/or executors/JVMs. These applications enjoy high throughput and low latency if they are served entirely from memory. However, actual estimation and memory allocation are difficult. When these applications cannot fit their working sets in real memory of their VMs/containers/executors, they suffer large performance loss due to excess page faults and thrashing. Even when unused host memory or unused remote memory are present in other VMs or containers and executors, these applications are unable to share those unused host/remote memory. Existing proposals focus on estimating working set size for accurate allocation, and increasing effective capacity of executors, but lack of desired transparency and efficiency.

This dissertation research takes a holistic approach to tackle the above problems from three dimensions. First, we present the design of *FastSwap*, a highly efficient shared memory paging facility. *FastSwap* dynamic shared memory management scheme can effectively utilize the shared memory across VMs through host coordination, with three original contributions. (1) *FastSwap* provides efficient support for multi-granularity compression of swap pages in both shared memory and disk swap devices. (2) *FastSwap* provides an adaptive scheme to flush the least recently swap-out pages to disk swap partition when shared memory swap partition reaches a pre-specified threshold and close to full. (3) *FastSwap*

provides batch swap-in optimizations. Our extensive experiments using big data analytics applications and benchmarks demonstrate that *FastSwap* offers up to two orders of magnitude performance improvements over existing memory swapping methods.

Second, we develop *XMemPod* for non-intrusive host/remote memory sharing and for improving performance of memory-intensive applications. It leverages the memory capacity of host machines and remote machines on the same cluster to provide on-demand, transparent and non-intrusive sharing of unused memory, effectively removing the performance degradation of big data and ML workloads due to transient or imbalanced memory pressure experienced on a host or in a cluster. We demonstrate the benefit of *XMemPod* design and the benefits of memory sharing via three optimizations: First, we provide elasticity, multi-granular compressibility and failure isolation on shared memory pages. Second, we implement hybrid swap-out for better utilization of host and remote shared memory. Third but not the last we support proactive swap-in from remote to host, from disk to host, and from host to guest, which improves paging-in operations significantly and opportunistically and shortens the performance recovery time of those applications under memory pressure. *XMemPod* is deployed on a virtualized RDMA cluster without any modifications to user applications and the OSes. Evaluated with multiple workloads on unmodified Spark, Apache Hadoop, Memcached, Redis and VoltDB, using *XMemPod*, throughputs of these applications improve by 11x to 612x over conventional OS disk swap facility, and by 1.7x to 14x over the existing representative remote memory paging system.

Third, we propose an efficient and elastic huge page management facility for memory intensive Bigdata and machine learning applications. Both computer hardware and operating systems provide support for huge pages. Modern computer hardware supports huge pages to handle hardware address translation overheads by providing thousands of entries in TLB for huge pages. Operating systems and hypervisors provide certain level of support for huge pages with best effort algorithms to address the access and management cost of memory page table for increasing DRAM capacity and growing memory footprint of Big-

data workloads. However, existing solutions in kernel support for huge pages are limited to spot fixes and some inherent fairness problems. We propose to take a methodical and principled approach to providing efficient, highly elastic, and yet transparent support of huge pages, aiming at improving utilization and access efficiency of memory pages.

CHAPTER 1
INTRODUCTION

Big data and latency-demanding applications [1, 2, 3, 4, 5] are typically deployed using the application deployment models, comprised of virtual machines (VMs), containers, and/or executors/JVMs. These applications enjoy high throughput and low latency if they are served entirely from memory. However, actual estimation and memory allocation are difficult. When these applications cannot fit their working sets in real memory of their VMs/containers/executors, they suffer large performance loss due to excess page faults and thrashing. Even when unused memory is present in other VMs/containers/executors on the same host or a remote node, these applications are unable to share those unused host/remote memory.

Existing research studied the above problems from two orthogonal dimensions: (i) estimating working set size for accurate resource allocation and (ii) increasing effective resource capacity for executors. Accurate memory allocation is hard as peak memory variations happen under different application types, workload inputs, data characteristics and traffic patterns. Applications often overestimate their requirements or attempt to allocate for peak usage [6, 7, 8, 9, 10], resulting in severely unbalanced memory usage across executors, underutilization on the host and across the cluster [11, 6, 12]. Instead, proposals for increasing effective memory capacity promote the allocation of global memory resource shared by all machines (VMs/containers) to increase their effective memory capacities. These proposals promote new architectures and new hardware design for memory disaggregation [13, 14, 15, 16, 17, 18], or new programming models [19, 20]. But they lack of desired transparency at OS, network stack, or application level, hindering their practical applicability. Recent efforts represented by Infiniswap [21] and Accelio nbdX [22] exploit RDMA networks for remote memory paging with transparency. However, they

adopt a flat architecture for disaggregated memory, fail to orchestrate host unused memory as the faster tier over the remote memory. Moreover, modern servers support terabytes of RAM and memory intensive applications able to take advantage of such large amount of memories are common. But, increased capacity means a significant challenge for address translation.

To tackle these challenges of memory resource utilization, this dissertation research is focused on the development of high performance, efficient, and transparent memory management framework for big data and latency-demanding applications.

1.1 Technical Challenges

We describe the technical challenges to build a high performance, efficient, and transparent memory management framework in more details as follows.

1.1.1 Transient Memory Usage Variation

Memory utilization imbalance and temporal usage variations are frequently observed in virtualized clouds [6, 23, 24, 25, 7, 8, 26, 27, 28], and production datacenters [29, 5, 30, 21, 11, 12]. One study on a Google 12K-machine cluster running a mixture of long and short-lived workloads reported around 50% memory utilization, stating "the gap between resource requests and average usage accounted for most of this difference" [9, 11]. Another study on two production datacenters (3,000-machine Facebook analytic cluster and 12,500-machine Google cluster) reports severe imbalance in memory utilization for more than 70% of the time across machines [21]. They demonstrated that there exists significant spatial imbalance (heterogeneous resource utilization across machines) and temporal imbalance (time varying resource usages per machine) of utilization for machines in cloud data center. These reports also show the potential opportunities of exploiting unused host/remote memory to speed up the performance of VM workloads whose working sets cannot fully fit into their allocated DRAM, and resorting only to disk I/O for persistence,

failure recovery and contingence.

1.1.2 Dynamic Memory Balancing

Existing research on scheduling and consolidating physical memory among VMs with single server environment centered on dynamic memory balancing, with Ballooning [31] as the most representative. Ballooning requires manual administration of when to trigger ballooning and how much ballooning is sufficient. Moreover, solely relying on ballooning, applications under memory pressure still suffer hefty performance degradation due to three types of delays: the timing delay of scheduling ballooning, the balloon driver delay for moving sufficient memory, and the time delay for applications to return to their peak performance. Proposals [32, 33, 34] devoted to periodic estimation of VM working set size. But, accurate working set prediction is hard under changing workloads [8, 35, 36]. In the cluster environment, distributed shared memory (DSM) was studied extensively [37, 38, 39, 19, 40, 41]. However, DSM suffers poor performance due to high communication overhead. Disaggregated memory has attracted much attention recently [42, 43, 17, 44, 15, 45]. Most proposals rely on new hardware architecture, new network protocols to cut down the communication cost. Some proposals show the benefit of leveraging RDMA technology [46, 47, 48, 49, 50, 51, 52, 48], such as remote storage for key-value stores[53, 54, 55, 56], distributed objects [57], swap pages [58, 59, 60, 61, 62, 21], object replication [63]. Most of these efforts lack of desired transparency. In virtualized clouds, multiple VMs run on the same host machine, each having its own memory allocation (typically equal amount). When a VM reaches its memory limit, it has two alternative low-latency memory expansion opportunities before paging to local disk swap partition. They are unused memory on local host and unused remote memory in the cluster. Note that even though SSD is much faster than HDD, it is still two orders of magnitude slower than fast interconnect like InfiniBand based RDMA [64]. At the same time, RDMA interconnect is several times slower than local memory access [65], making latency-sensitive prioritization critical for

efficient disaggregated memory orchestration.

1.1.3 Memory Address Translation

A fundamental property of virtual memory is that the CPU references a virtual address that is translated via a combination of software and hardware to a physical address. This allows data only to be paged into memory on demand improving memory utilization. Use of virtual memory is pervasive but this indirection is not without cost. Due to translation, a virtual memory reference necessitates multiple accesses to physical memory, multiplying the cost of an ordinary memory reference by a factor depending on the page table format. To cut the costs associated with translation, virtual memory implementations take advantage of the principal of locality by storing recent translations in a cache called the Translation lookaside buffer (TLB). It is a part of memory management unit (MMU). The TLB stores the recent translations of virtual memory to physical memory and can be called an address translation cache. Once an instruction asks the processor to do memory operation on a virtual address, the processor first checks to see whether the TLB contains an entry for that virtual address. If the requested address is present in the TLB, the search yields a match quickly and the retrieved physical address can be used to access memory. This is called a TLB hit. If the requested address is not in the TLB, it is a miss, and the translation proceeds by looking up the page table in a process called a page walk. The page walk is expensive when compared to the processor speed, as it involves reading the contents of multiple memory locations and using them to compute the physical address. Modern servers support terabytes of RAM and memory intensive applications able to take advantage of such large amount of memories are common. But, increased capacity means a significant challenge for address translation.

1.2 Dissertation Scope and Contributions

In this dissertation, we propose solutions to build a high performance, efficient, and transparent memory management framework. We achieve our goal from the following aspects:

First, we present the design and the results of a measurement study on in-memory key-value system with three original contributions. (1) We perform a comparative study on two most popular and most efficient in-memory key-value systems, Redis [2] and Memcached [1], in terms of the in-memory data placement and processing techniques. This includes internal data structure in memory, fragmentation, caching, read and write operation performance. (2) We provide experimental evaluation and analysis on the performance impact of different in-memory data structures on different workloads, including a comparison on commonality and difference in the design of Redis and Memcached with respect to memory efficiency. (3) We attempt to answer a number of challenging and yet most frequently asked questions regarding in-memory key-value systems through extensive and in-depth experimentation and measurements. Example questions include but not limited to: What types of data structures are effective for in-memory key value systems? How do in-memory key-value systems respond to the big data workloads, which exceeds the capacity of physical memory or the pre-configured size of in-memory data structures? What types of persistence models are effective for maintaining persistency of in-memory key value systems? And why do different in-memory key value systems have different throughput performance and what types of overheads are the key performance indicators? To the best of our knowledge, this is the first in-depth comparative measurement study on in-memory key-value systems.

Second, we design *FastSwap*, a host-coordinated shared memory swapper with multi-granularity compression on memory swap pages. The design of *FastSwap* is original in three aspects: (1) By creating a shared memory based swap area between the host and its VMs, *FastSwap* provides an efficient and transparent memory sharing mechanism. This

shared memory mechanism can effectively leverage idle memory present in the host or the other VMs on the host and speed up the VM memory swapping performance by minimizing or avoiding the high overhead of disk I/O for memory swapping on guest VMs. (2) By implementing a compressed swap page table as an efficient index structure to enable fast lookup of compressed pages in four granularity groups from its swap partitions, *FastSwap* improves both memory swap-out and swap-in performance. (3) *FastSwap* provides a hybrid swap-out scheme to handle the situation of limited shared memory for VM memory paging and a two-level proactive batch swap-in facility to speed up the performance of VM memory swap-in operations in the presence of sufficient shared memory.

Third, we develop *XMemPod*, a disaggregated memory orchestration system that virtualizes cluster wide memory to scale data intensive, large memory workloads in virtualized clouds. No modification is required for guest OSes, host OSes, or the hypervisor to deploy *XMemPod*, which makes it more practical and flexible in real cloud environment. The goal of *XMemPod* is to accelerate bigdata and machine learning workloads by virualizing two types of cluster-wide external memory. First, the available memory from other VMs on the same node due to inter-VM memory usage imbalance. Second, The available remote memory in the cluster through RDMA registered memory allocation. We make three novel contributions in this work. (1) By providing efficient and transparent sharing of unused memory that is disaggregated across VMs on the same host or in the cluster, *XMemPod* provides the ability of dynamically expanding the memory capacity of the virtual machines (VMs) under high memory pressure. (2) By providing a hierarchical memory expansion and sharing framework, *XMemPod* enables memory intensive workloads on a VM to expand its memory demand over virtualized host memory first, and remote memory next, before resorting to external disk. (3) To further improve the utilization and access latency of disaggregated memory, *XMemPod* provides a suite of optimization techniques. It is deployed on a virtualized RDMA cluster without any modifications to user applications and the OSes.

Fourth, we introduce *Xpage*, a memory management framework that effectively mitigates fragmentation and makes huge page benefits accessible to applications even in stressful conditions. This work is motivated by real problems that memory intensive software companies face when dealing with large amount of data. Unlike Linux and Ingens, which demotes huge pages to base pages once fragmentation occurs, *Xpage* never splits huge pages. Instead, it tracks allocated/unallocated memory regions and compacts all memory regions in memory allocator level by placing them carefully within huge page boundaries. *Xpage* is implemented with three original contributions: (1) *Xpage* interacts with both application (`malloc/free`) and kernel (`brk/mmap`) to get a global view of memory layout, which is parsed and analyzed for deciding the degree of internal fragmentation. (2) To ensure fairness and security, *Xpage* deals with internal fragmentation within each process's memory space and use each process's CPU time to execute tasks. It is a run-time library and completed isolated from other running process memory space. (3) By avoiding the kernel space overhead and complexity, *Xpage* is the first memory management framework that effectively deals with huge page fragmentation problem in memory allocator level. It is a redesign that brings performance and memory saving to memory intensive applications with dynamic memory behavior.

1.3 Dissertation Organization

This dissertation consists of several chapters and each chapter addresses one or more of the problems described above. In each chapter, we introduce the background of the problem being addressed, describe related work, and present our solution techniques followed by experimental evaluation. This dissertation is organized as follows.

In Chapter 2, we present a in-depth measurement study on critical performance and design properties of in-memory key-value systems, including the general infrastructure design, the use of different internal data structures, different persistency models, different policies for memory allocators, overhead of bulk insertion of large datasets, swapping

overhead, dictionary expansion overhead, space overhead, caching overhead, read/write operation performance, and fragmentation ratio.

In Chapter 3, we propose a efficient shared-memory based memory paging service, *FastSwap*, which improves VM memory swapping performance by leveraging idle host memory and redirecting the VM swapping traffic to the host-guest compressed shared memory swap area. It consists of four core functional components: dynamic shared memory management, hybrid swap-out, two level proactive swap-in, and multi-granularity compression of shared memory.

In Chapter 4, we introduce a disaggregated memory orchestration system, *XMemPod*, which virtualizes cluster wide memory to scale data intensive and large memory workloads in virtualized cloud. *XMemPod* consists of four core components: the disaggregated memory paging service manager, the shared memory manager, the remote memory manager and the disk swap partition manger. They coordinate closely to provide efficient virtualization of cluster wide memory and hierarchical orchestration of disaggregated memory sharing capabilities.

In Chapter 5, we develop huge page management framework, *Xpage*, which efficiently mitigates fragmentation and brings performance and memory saving to memory intensive applications with dynamic memory behavior. *Xpage* consists of three core components: the object address table, the fragmentation monitor, and the huge page manager which interacts with three sub managers that are tcache manager, non-full run manager, and full run manager.

In Chapter 6, we summarize the main contribution of this dissertation and discuss our future research directions.

1.4 Thesis Statement

The thesis of this dissertation is that the tools and techniques developed make it possible to dynamically instrument an already-running commodity operating system kernel in

a fine-grained manner to optimize memory resource utilization in both local host and remote nodes in a cluster. This technology can be usefully applied to kernel performance measurement and run-time performance optimization.

Concretely, this dissertation research has made three unique contributions to the general field of memory virtualization in Big data clouds. (1) A transparent and elastic shared memory system for memory sharing across VMs on the same host (2) A transparent and elastic host and remote memory virtualization system for disaggregated memory sharing across VMs on local nodes and remote nodes in the cluster. (3) A transparent and efficient huge page management system for boosting execution performance of big data applications.

FastSwap, *XMemPod*, and *Xpage* are open sourced on https://github.com/git-disl.

CHAPTER 2

PERFORMANCE ANALYSIS OF IN-MEMORY KEY-VALUE SYSTEMS

The rapid advance of memory technology and the continued price drop of memory chips, compound with big data processing demand, have accelerated the development and deployment of in-memory key value systems. In this chapter, we present an in-depth measurement study on in-memory key-value systems. We examine the in-memory data placement and processing techniques, including in-memory data structures, caching, performance of read and write operations, the memory efficiency of different in-memory data structures and their impact on the performance of big data workloads, memory efficiency. Based on the analysis of our measurement results, we attempt to answer a number of challenging and yet most frequently asked questions regarding in-memory key-value systems through extensive and in-depth experimentation and measurements, such as how do in-memory key-value systems respond to the big data workloads, which exceeds the capacity of physical memory or the pre-configured size of in-memory data structures? How do in-memory key value systems maintain persistency and manage the overhead of supporting persistency? why do different in-memory key-value systems show different throughput performance? and what types of overheads are the key performance indicators? We conjecture that this study will benefit both consumer and providers of big data services and help big data system designers and users to make more informed decision on configurations and management of key-value systems and on parameter turning for speeding up the execution of their big data applications.

2.1 Introduction

As big data penetrating both business enterprises and scientific computing, in-memory key-value systems are gaining increased popularity in recent years. For example, in-memory

key-value store is a crucial building block for many industrial big data systems and applications, including Facebook, Twitter, Google, Amazon EC2, for providing high performance big data services. In contrast to disk-based key-value systems, such as HBase [66], Mongo [67], Cassandra [68], in-memory key-value systems, such as Redis [2], Memcached [1], rely on memory to manage all their data by utilizing virtual memory facility when the dataset to be loaded and processed is beyond the capacity of the physical memory. In memory key-value systems are faster than disk-optimized key-value systems for a number of reasons: minimized disk I/O cost, simpler internal optimization algorithm and data structure, and fewer CPU instructions. For example, reading 1MB data sequentially from memory takes 0.25ms, while reading the same amount of data sequentially from disk takes 20ms, which is 80$\times$ slower than memory [69]. Thus, the ability to keep big data in-memory for processing and performing data analytics can have huge performance advantages.

Research on in-memory systems can be dated back to the 80s [70, 71]. The recent advances and price reduction in computer hardware and memory technology have further fueled the growth of in-memory computing systems. In-memory key-value systems become one of the most prominent emerging big data technologies and an attractive alternative to disk-optimized key-value systems for big data processing and analytics [2, 1, 72, 73, 69]. However, there is no in-depth performance study with quantitative and qualitative analysis on in-memory key-value systems in terms of the design principle, the effectiveness of memory utilization, the key performance indicators for in-memory computing, and the impact of in-memory systems on big data applications.

In this chapter, we present the design and the results of a measurement study on in-memory key-value systems with two objectives: First, we plan to perform a comparative study on two most popular and most efficient in-memory key-value systems, Redis [2] and Memcached [1], in terms of the in-memory data placement and processing techniques. This includes internal data structure in memory, fragmentation, caching, read and write operation performance. Second, we plan to provide experimental evaluation and analysis

on the performance impact of different in-memory data structures on different workloads, including a comparison on commonality and difference in the design of Redis and Memcached with respect to memory efficiency. As a result of this study, we attempt to answer a number of challenging and yet most frequently asked questions regarding in-memory key-value systems through extensive and in-depth experimentation and measurements. Example questions include but not limited to: (1) What types of data structures are effective for in-memory key value systems? (2) How do in-memory key-value systems respond to the big data workloads, which exceeds the capacity of physical memory or the pre-configured size of in-memory data structures? (3) What types of persistence models are effective for maintaining persistency of in-memory key value systems? And (4) Why do different in-memory key value systems have different throughput performance and what types of overheads are the key performance indicators? To the best of our knowledge, this is the first in-depth comparative measurement study on in-memory key-value systems. We conjecture that this study will benefit both consumer and providers of big data services. The results of this study can help big data system designers and users make informed decision on configurations and management of key-value systems and on performance turning for speeding up the execution of their big data applications.

Finally, neither study considers the system that we include, Redis.

The remaining of this chapter is structured as follows: Section 5.7 gives an overview of in-memory systems. Section 2.3 presents the design and methodology for our measurement study. Section 2.4 describes the detailed measurement results and performance and overhead analysis. We provide related work in Section 4.7 and conclude the chapter in Section 4.8.

2.2 Overview

In this section, we provide a brief overview of general in-memory key-value systems, and a review of Redis [2] and Memcached [1] with respect to the set of target features evaluated

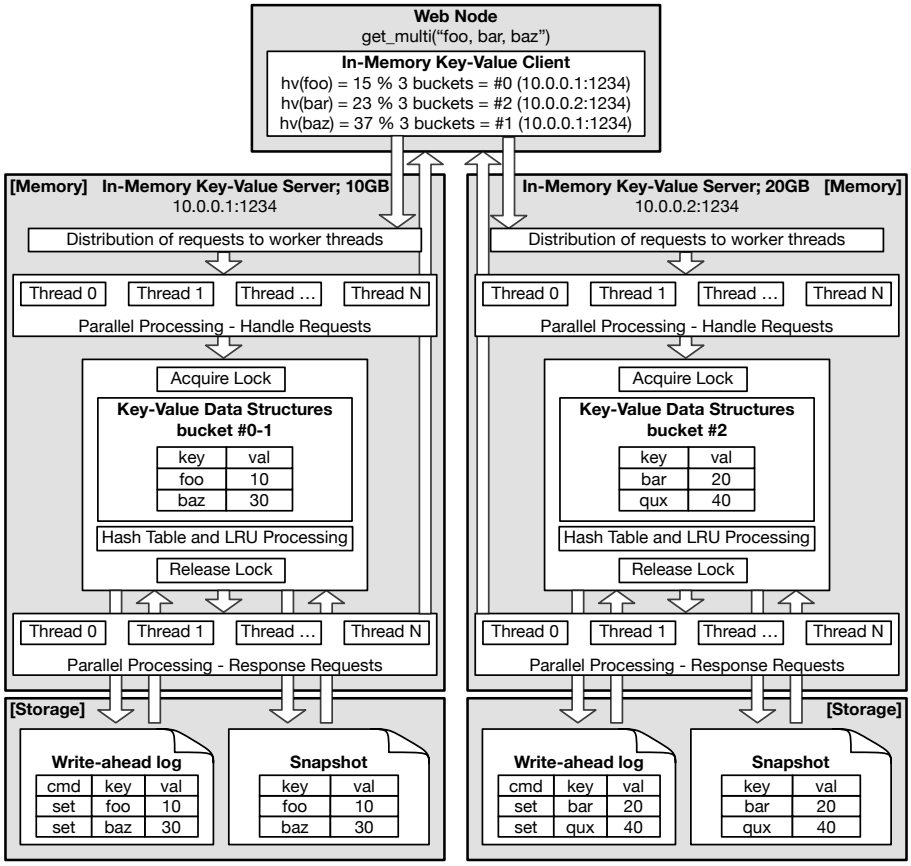


Figure 2.1: Typical Infrastructure of In-Memory System

in our measurement study, including the in-memory data structures and some key memory optimization features.

2.2.1 In-Memory Key-Value Systems

The key-value systems achieve high performance through three common systems-level techniques: (i) *parallel request handling*, (ii) *efficient network I/O*, and (iii) *concurrent data access*. Concretely, through hashing, key-value pairs can be distributed almost evenly to each compute node of a key-value system cluster. Client requests are served concurrently by accessing different servers with minimum performance interference. To deal with the latency of network I/Os, key-value systems employ pipeline to support asynchronous request processing. It allows multiple requests to be sent to the server without waiting for the replies, and batch deliver the replies in a single step. In addition to utilize multi-

ple CPU cores and multithreading for concurrent data access, key-value systems maintains data integrity using mutexes, optimistic locking, or lock-free data structure.

Unlike disk-optimized key-value systems, such as HBase [66], Cassandra [68], in-memory key-value systems rely on internal memory data structures to manage all application data and intelligently utilize virtual memory facility to handle large datasets that exceed the size of allocated physical memory. In-memory key-value systems also employ logging and snapshot techniques to provide persistency support. Figure 2.1 shows a typical architecture of an in-memory key-value system.

The in-memory key-value systems differ from one another in terms of their memory management strategies, such as memory allocation, virtual memory facility, concurrent access methods, and their persistency implementation models. For example, an in-memory key-value system typically needs to allocate and deallocate portions of memory of varying sizes many times during execution. The high frequency of performing these memory operations makes their performance significantly important. Inadequate use of internal data structure to host data in memory may detriment the memory efficiency and thus the performance of in-memory key-value systems. Frequent memory allocation/deallocation operations can also cause undesired memory fragmentations. Another important performance component is efficient virtual memory facility, which is critical for in-memory key-value systems to handle large datasets that exceed the allocated physical DRAM. Thus, some in-memory key-value systems use their own virtual memory management policy, such as Redis older versions before version 2.4, whereas others fully rely on memory reclamation mechanism of the operating system (OS). To handle the volatility of DRAM memory and data loss due to reboot or power outage, in-memory key-value systems need to provide durability support by either implementing some persistent models, such as snapshot, transaction logging, or deploying non-volatile random access memory or other high availability solutions, as illustrated at the bottom part of Figure 2.1. However, the persistency model also introduces another source of system overhead. Redis [2] and Memcached [1] are the

two most popular in-memory key-value systems, because they are super fast compared to other existing key-value systems.

2.2.2 Redis Overview

Redis is an open source, in-memory data structure store, used as database, cache and message broker. It is implemented in C and adopts jemalloc [74] as its memory allocator since version 2.4, although its early versions use glibc [75] as the default memory allocator. The quality of a memory allocator is determined by how effectively it can reduce the memory usage for storing the same amount of dataset and alleviate the impact of fragmentation. Redis supports four sets of complex data structures to allow application developers to select the best memory data structures based on their dataset sizes and workload characteristics: (1) *Lists* - collections of string elements sorted by the order of insertion; (2) *Sets* - collections of unique, unsorted string elements; (3) *SortedSets* - similar to Sets but every string element is also associated with a floating number value, called score. The elements are sorted by their score; and (4) *Hashes* - which are maps composed of fields associated with values. For each of the four types of data structure, two implementations are provided, each with its unique characteristics. For example, linked list and ziplist are two implementations of the list data structure. Previously, Redis had its own virtual memory facility to handle swap-in and swap-out bypassing the OS virtual memory management. The latest version 4.2 has changed to provide the in-memory dataset to host all data in DRAM and let OS handle all swap-in and swap-out paging requests when the dataset size exceeds the allocated physical memory. We will analyze the reasons for this change in our measurement study. Redis also supports five different persistency models: *nosave*, *snapshot*, *AOF-No*, *AOF-Everysec*, and *AOF-Always*. *nosave* denotes no support for persistency. *snapshot* periodically takes snapshots of all the working dataset hosted in memory and then dumps the snapshots into persistence storage system stored as the latest snapshot files. The three AOF models are log based and refer to log-immediate, log-periodical and log-deferred, with Log-Periodical as

Table 2.1: Number of Objects per Entry and Space Overhead per Entry (bytes) - 1
(m:key length, n:value length, k:field length, h:# fields, g:# elements, L:*List*, S:*Set*, SS:*SortedSet*, H:*Hash*)

Struct			dict	dictEntry	dictht	redisObject	sdshdr
L	ziplist	min	0	24	0	32	8+m
		max	0	24	0	32	8+m
	linkedlist	min	0	24	0	32+16g	8+m+8g+ng
		max	0	24	0	32+16g	8+m+8g+ng
S	intset	min	0	24	0	32	8+m
		max	0	24	0	32	8+m
	hashtable	min	96	24+24g	32	32+16g	8+m+8g+ng
		max	96	24+24g	64	32+16g	8+m+8g+ng
SS	ziplist	min	0	24	0	32	8+m
		max	0	24	0	32	8+m
	skiplist	min	96	24+24g	32	32+16g	8+m+8g+ng
		max	96	24+24g	64	32+16g	8+m+8g+ng
H	ziplist	min	0	24	0	32	8+m
		max	0	24	0	32	8+m
	hashtable	min	96	24+24g	32	32+32g	8+m+16g+kg+ng
		max	96	24+24g	64	32+32g	8+m+16g+kg+ng

Table 2.2: Number of Objects per Entry and Space Overhead per Entry (bytes) - 2
(m:key length, n:value length, k:field length, h:# fields, g:# elements, L:*List*, S:*Set*, SS:*SortedSet*, H:*Hash*)

Struct			ziplistlayout	list	listNode	intset	zset	zskiplist	zskiplistNode
L	ziplist	min	11+2g+ng	0	0	0	0	0	0
		max	11+10g+ng	0	0	0	0	0	0
	linkedlist	min	0	48	24g	0	0	0	0
		max	0	48	24g	0	0	0	0
S	intset	min	0	0	0	8+2g	0	0	0
		max	0	0	0	8+2g	0	0	0
	hashtable	min	0	0	0	0	0	0	0
		max	0	0	0	0	0	0	0
SS	ziplist	min	11+5g+ng	0	0	0	0	0	0
		max	11+21g+ng	0	0	0	0	0	0
	skiplist	min	0	0	0	0	16	32	24+24g
		max	0	0	0	0	16	32	24+24g
H	ziplist	min	11+4g+ng+kg	0	0	0	0	0	0
		max	11+20g+ng+kg	0	0	0	0	0	0
	hashtable	min	0	0	0	0	0	0	0
		max	0	0	0	0	0	0	0

the tradeoff of persistence and write performance between log-immediate and log-deferred.

2.2.3 In-Memory Key-Value Data Structures

Redis lists are simply lists of strings, sorted by insertion order. It is possible to add elements to a Redis list by pushing new elements onto the head (on the left) or the tail (on the right) of the list. The main feature of Redis list is the support for constant time insertion and deletion of elements near the head and the tail, even with many millions of inserted items. Accessing elements is very fast near the head or tail of the list but is slow when accessing

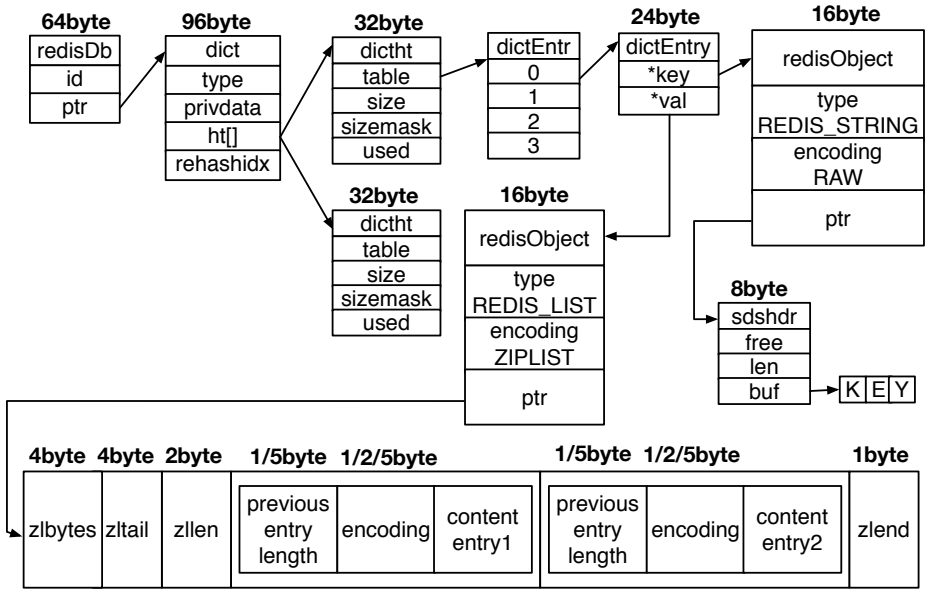


Figure 2.2: *Ziplist* Data Structure (List, SortedSet, HashTable)

the middle of a very big list, with an O(N) operation. Figure 2.2 and Figure 2.3 show the two implementations of list data structure: unlike *linkedlist*, *ziplist* uses continuous memory space, has the constant time complexity O(1) with compact memory structure, but it pays the cost of memory resizing and movement due to insertion and deletion of entries. In contrast, with the flexible structure, *linkedlist* is more efficient for insertion and deletion of entries than *ziplist*. Given that all data are linked by pointers of *listnote*, *linkedlist* scales nicely for increasing sizes of raw datasets, but it suffers from larger memory consumption. Table 2.2 shows that the maximum memory consumption of one entry of *ziplist* is 75+m+(10+n)×g, and an entry of *linkedlist* is 112+m+(48+n)×g, 37+38×q additional bytes than an entry of *ziplist* and q is the number of elements in an entry.

Redis sets are an unordered collection of Strings. It is possible to add, remove, and read elements in O(1) for *hashtable* implementation (constant time regardless of the number of elements contained inside the set). Redis sets have the desirable property of not allowing repeated members. Adding the same element multiple times will result in a set having a single copy of this element. An *intset* is a sorted array of integers as shown in Figure 2.4. A binary search algorithm is used to find an element with the time complexity of O(log

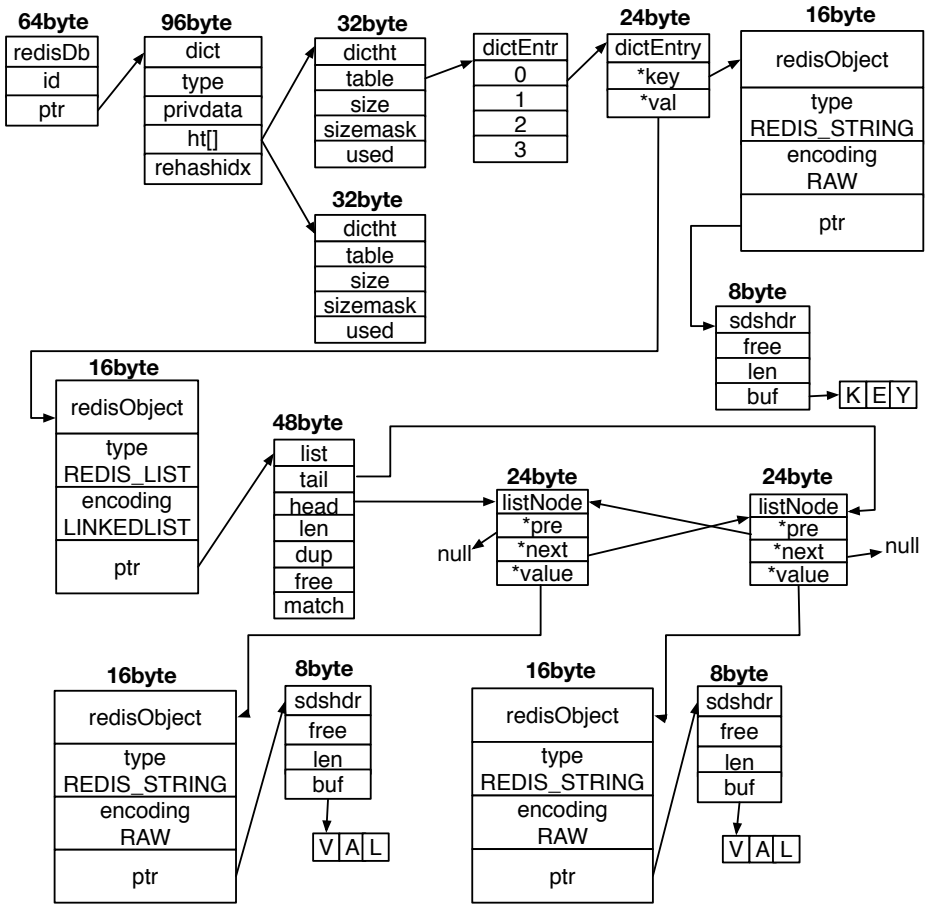


Figure 2.3: List - *Linkedlist* Data Structure

N). Adding new integers may require a memory reallocation, which can be expensive for large integer arrays. In contrast, the *hashtable* implementation simply uses hash function to construct set structure as shown in Figure 2.5. It offers good performance for large dataset at the cost of much more memory space than *intset*. Table 2.2 shows that the maximum memory consumption of one entry of *intset* is only 72+m+2×g, compared to 224+m+(48+n)×g for an entry of *hashtable*, 152+46×q additional bytes per entry.

Redis sortedsets are non repeating collections of Strings. The difference from the Redis sets is that every member of a *sortedset* is associated with a score such that the sorted set is ordered accordingly from the smallest to the largest score. Different members may have the same scores. Adding, removing, or updating elements can be done in a very fast way with the time complexity O(log N), and N is the number of elements. Similarly, getting

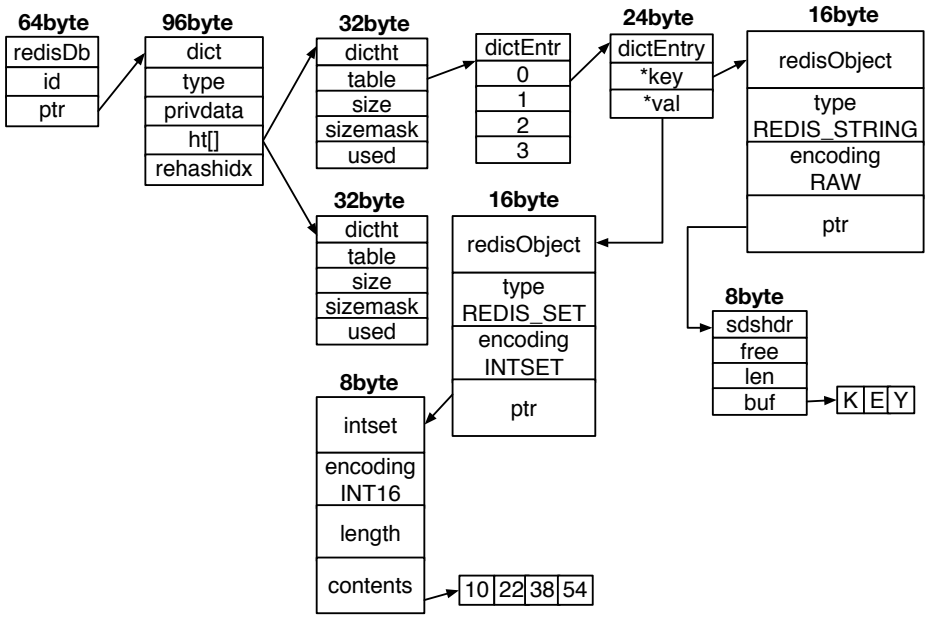


Figure 2.4: Set - *Intset* Data Structure

ranges of elements by score or by rank (position) is fast, and so does the access to the middle of a sorted set. One can use *sortedset* as a smart list of non repeating elements to quickly access everything you need: elements in order, fast existence test, fast access to elements in the middle. *Sortedset* has two implementations: *ziplist* and *skiplist*. *ziplist* inevitably suffers from memory reallocation and movement (see the *ziplist* data structure in Figure 2.6). *Ziplist* keeps all entries sorted in a continuous memory space, and almost every add/remove operation incurs O(log N) search time and O(N) memory movement time, which is non-negligible overhead. *ziplist* by design trade its performance for better memory usage and cache line locality. Alternatively, *skiplist* has diametrically opposite design as shown in Figure 2.7: It creates huge number of objects for better search and update performance. Table 2.2 shows that the maximum memory consumption of one entry of *ziplist* is 75+m+(21+n)×g, compared to 296+m+(72+n)×g for an entry of *skiplist*, 221+51×q more additional bytes per entry.

Redis Hashes are maps between string fields and string values for representing objects of multiple features (attributes). *Ziplist* and *hashtable* are the two implementations that

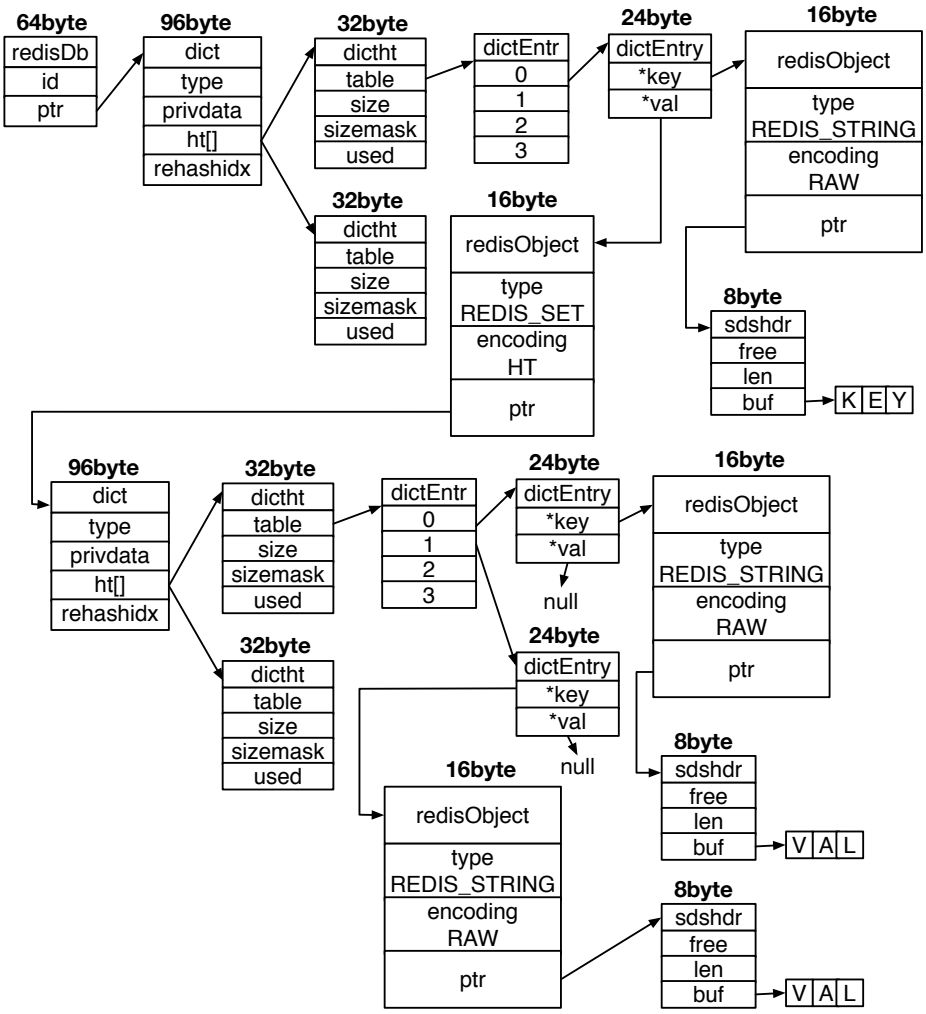


Figure 2.5: Set - *Hashtable* Data Structure

we've discussed previously.

2.2.4 Memcached Overview

Memcached is a general-purpose distributed memory caching system [1]. It is often used to speed up dynamic database-driven websites by caching data and objects in DRAM to improve the performance of reading from an external data source.

Data items in Memcachd are organized in the form of key and value pair with meta-data. Given the varying sizes of data items, a naive memory allocation scheme could result in significant memory fragmentation problem. Memcached addresses this problem by

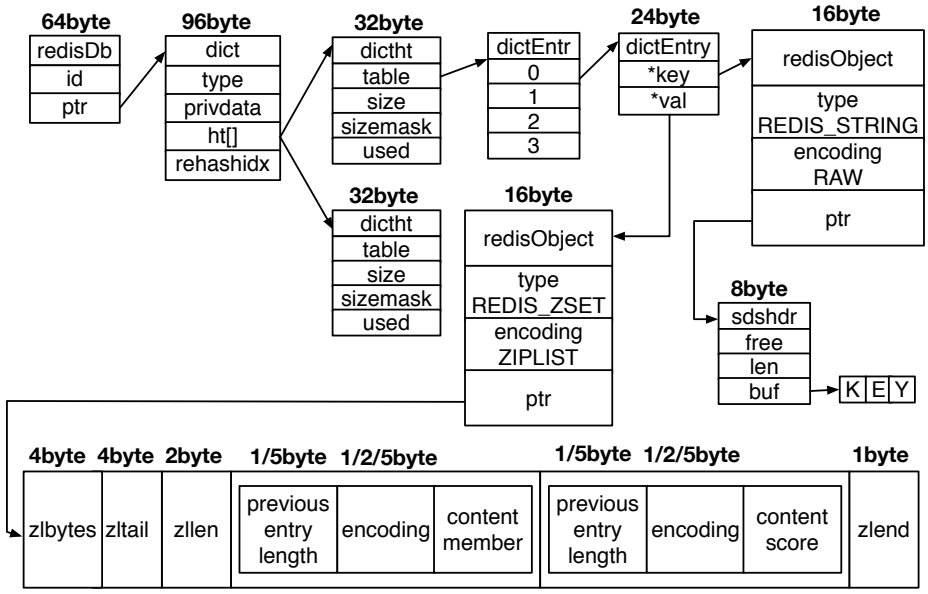


Figure 2.6: SortedSet - *Ziplist* Data Structure

combining glibc with slab allocator [76], since glibc alone cannot handle the memory fragmentation problems. In Memcachd, the memory is divided into 1MB pages. Each page is assigned to a slab class, and then is broken down into chunks of a specific size for the given slab class. Upon the arrival of new data items, a search for the slab class of the best fit in terms of memory utilization is initiated. If this search fails, a new slab of the class is allocated from the heap and otherwise the new item is pushed into a chunk in the chosen slab class. Similarly, when an item is removed from the cache, its space is returned to the appropriate slab, rather than the heap. Memory is allocated to slab classes based on the initial workload and its item sizes until the heap is exhausted. Therefore, if the workload characteristics change significantly after the initial phase, the slab allocation may not be appropriate for the workload, which can results in memory underutilization. For example, if the total of data entry with key + value + metadata is 50 bytes, it will go into the slab class 1 of size 96 bytes by default, resulting in an overhead loss of 46 bytes. Memcached is by design a memory caching system and thus very simple compared with Redis, since it only supports key-value pairs without sorting or complex data structures. Also it does not support any persistency model like Redis does, No failure and data loss recovery in

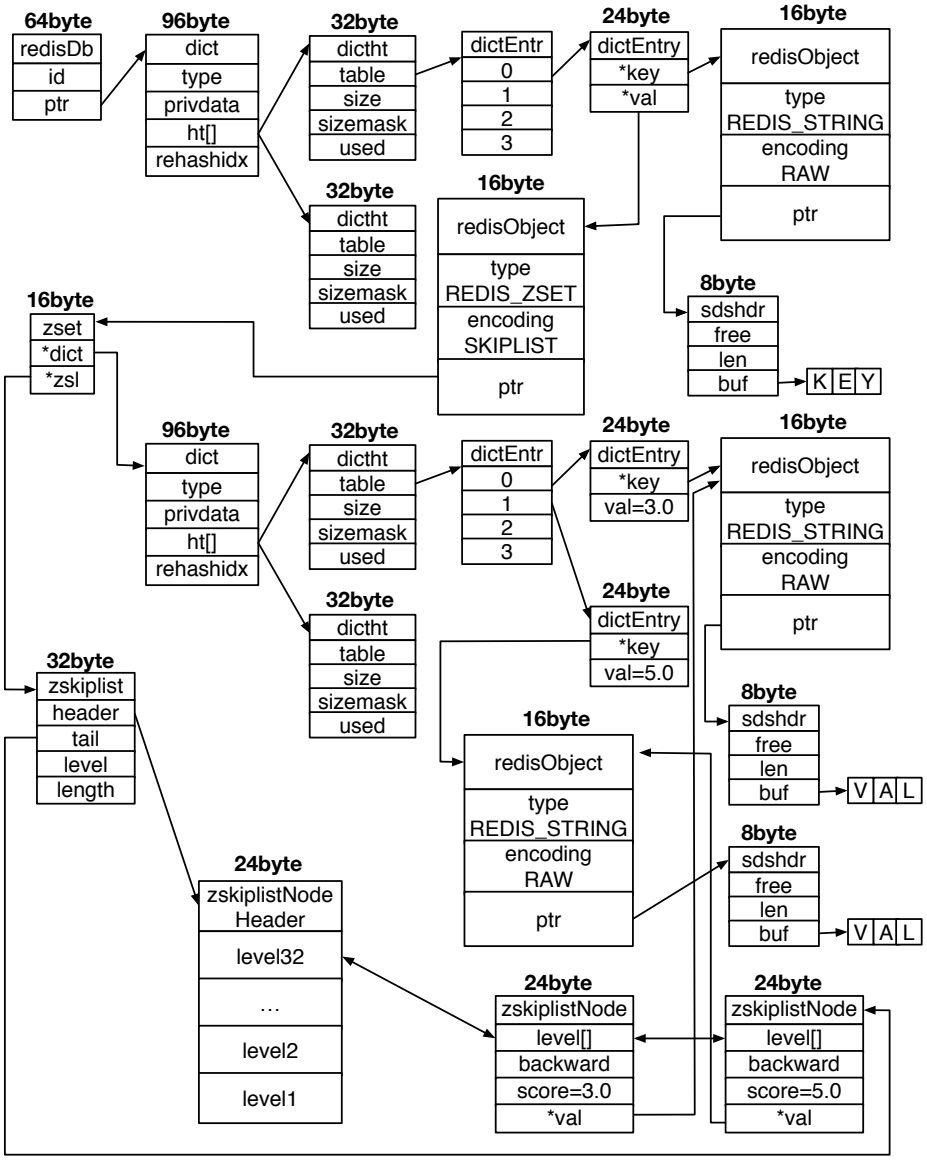


Figure 2.7: SortedSet - *Skiplist* Data Structure

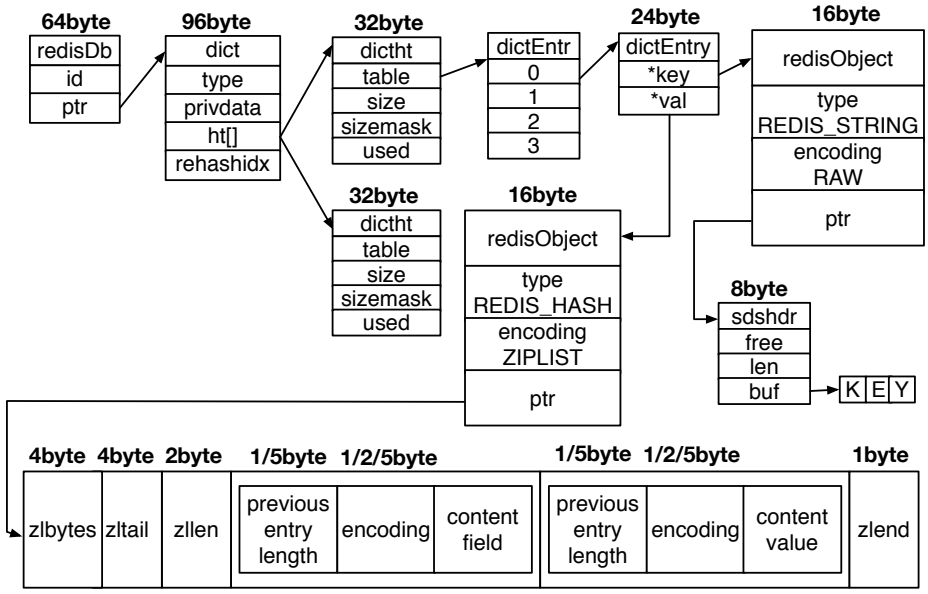


Figure 2.8: Hash - *Ziplist* Data Structure

Memcached due to reboot and power outage.

The workflow of storing a key-value entry is as follows: (i) Compute hash code of the key. (ii) Acquire corresponding bucket lock. (iii) Find the entry with the searched key in the bucket table; if present, updates the entry accordingly. (iv) Otherwise compute the total entry size with key + value + metadata. (v) Determine the best-fit slab class by looking for the closest ceiling entry size based on the given entry size. (vi) By examining the last entries in the LRU link for this slab class, searching for expired entries. After each unsuccessful visit, try to allocate the new entry using slab allocator. The search for placement of a new entry to a slab class continues until all new entries are hosted in memory or Memcached failed to allocate the entry because Memcached is running out of available memory.

In Memcached, its primary storage algorithm is a hash table, the size of which is power of two. Bucket index is determined by hash (key) modes the hash table size. Collision is resolved via separate chaining and new entries are inserted in the front of the collision chains. Figure 2.10 shows the internal data structure of Memcachd, where hash links put all key-value pairs with the same hash result on key component together using pointer *h_next. For search operation, Memcached needs to iterate a certain hash link until finding

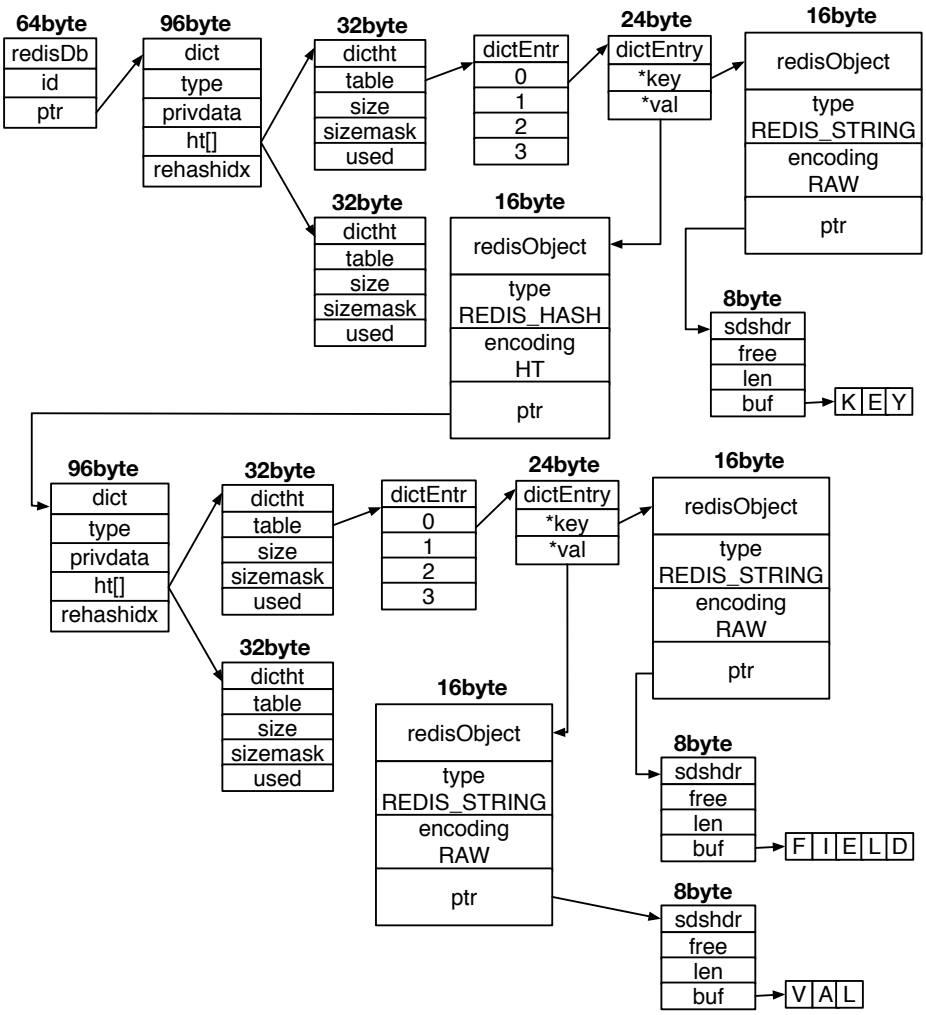


Figure 2.9: Hash - *Hashtable* Data Structure

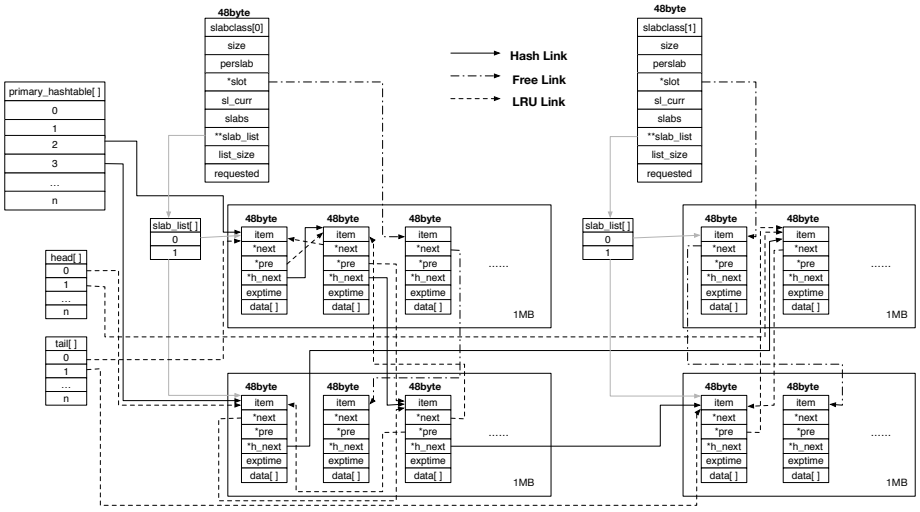


Figure 2.10: Memcached Internal Data Structure

the correct item. Hash table expansion is realized by allocation of the whole new table of double size and iteratively moving entries from the old table to the new in background. The *next and *prev pointers in Figure 2.10 serve different purposes: (1) When the entry is alive, these pointers are used to build doubly-linked list of LRU entries of some entry size class. (2) When the entry is free after just being allocated by memory allocator, or just performed a deletion operation, it will participate a doubly-linked list of free entries. During entry insertion, if an expired entry is found, it is unlinked from the old hash table bucket and repositioned in the LRU list; Otherwise, the newly allocated one is unlinked from the slabclass free entry list and inserted in front of the LRU chain. The overhead of allocating one entry in Memcached is approximate 72 bytes, including hash table pointers, *next and *prev pointers, *h_next pointer, access time, expiration time, value size, reference count, key size, flags-and-length string size, entry size class id, and flags.

Memcached stores key and value pairs along with entry structure within a single allocation, reducing memory overhead, improving access locality, lowering memory allocation pressure (frequency), and reducing fragmentation. However, this may also cause some problems: First, it is not straightforward to extend Memcached to support diverse data structures like Redis, because this fundamental design constrains its extension. Second, with slab allocator, although Memcached no longer needs to allocate memory as frequently as Redis, the use of slab allocator incurs more computation on memory allocation and leads to a more rigid memory allocation policy.

2.3 Methodology

In this section we describe the workload design, experimental platform setup, the evaluation models, and the measurement metrics used for our measurement study.

2.3.1 Workload Design

YCSB [77] is a popular workload generator for disk-resident key-value systems. Given that in-memory key-value systems are emerging in recent years, only a few benchmarking tools support both Redis and Memcached, and unfortunately YCSB does not. In our measurement study, all workloads are generated by *Memtier_benchmark* [78] produced by RedisLab, or GIT_KVWR (a Georgia tech key-value workload generator for Redis), developed by authors using Jedis interacting with Redis. Although Redis provides manifold data structure options (recall Section 2), none of the benchmarks, including *Memtier_benchmark* can create workloads to measure the performance for all of them. To address the problem that *Memtier_benchmark* only supports simple key-value workloads, we develop GIT_KVWR in order to conduct in-depth measurements on memory efficiency and performance impact of different internal data structures. The detailed settings are as follows: (1) *List*: Each record contains 50 entries, the length of each entry is 20B. *rpush* is adopted as insertion operation. (2) *Set*: Each record contains 50 entries, the length of each entry is 20B. *sadd* is adopted as insertion operation. (3) *SortedSet*: Each record contains 50 value-score pairs, the length of each value is 20B and the score is a double variable. *zadd* is adopted as insertion operation. (4) *Hash*: Each record contains 50 fields, each field contains value with 20B. *hset* is adopted as insertion operation. We want to examine the impact of different data structures, different record sizes, and different dataset sizes on the performance of in-memory key-value systems.

Memtier_benchmark is used to generate two types of workloads: (1) each record is 100×10 bytes and (2) each record of 1000×10 bytes. The first set of experiments uses the record size of 1KB and the second set of experiments uses the record size of 10KB. For both sets of experiments, *Memtier_benchmark* is used to generate 5GB and 10GB workloads, corresponding to 5,000,000$\times$1KB records and 10,000,000$\times$1KB records respectively.

GIT_KVWR is designed for generating workloads to measure the performance of the manifold data structure options in Redis, since none of the existing benchmarks can create

workloads for all of them.

Experimental Platform Setup. All experiments were performed on a server platform with 3.20GHz Intel Core-i5-4460 CPU (4 cores and 4 hyperthreads), which has 32KB L1d caches, 32KB L1i caches, 256KB L2 cache, and 6144KB L3 caches, 16GB 1600MHz DDR3 memory, and a Samsung SATA-3 250 GB SSD. Ubuntu 14.04 with Linux kernel 3.19.0-25-generic is installed. We use Redis 3.0.5 standalone mode with default settings, and Memcached 1.4.25 with the configuration of maximum 1024 connections and maximum 20480MB (20GB) memory allocation.

2.3.2 Evaluation Models

Our evaluation examines four types of performance overheads commonly observed over in-memory key-value systems: (i) performance overheads of handling large scale datasets, (ii) performance overheads of supporting different persistency models, (iii) performance overheads of supporting different in-memory data structures.

Measuring the overheads of handling large scale datasets. We use the bulk insertion workloads for this type of overhead measurement. For the 5GB (light load) and 10GB (heavy load) workloads generated by *Memtier_benchmark*, we track the throughput and fine-grained CPU and memory consumptions for three types of activities (user-level, kernel system level, I/O level) during insertion phase and generate fine-grained system performance statistics using *SYSSTAT* [79] and *Perf* [80]. Moreover, we also examine the memory swapping overheads during bulk insertion of 10GB workloads by allocating only 8GB physical memory. We examine both swap-in and swap-out activities and compare the physical memory utilization with the amount of swapping memory pages.

Measuring the overheads of different persistency models. Snapshot and logging are the two popular persistency models. Although it is widely recognized that supporting persistency can introduce performance overheads for in-memory key-value systems, very few studies the efficiency of different persistency models and how they impact on the perfor-

mance of different workloads and how they compare with the scenarios of no persistency support in terms of CPU and memory consumptions. We measure *nosave* in Redis as no-persistency support scenario, *Snapshot*, and three different logging scenarios: *AOF-No* for log-deferred, *AOF-Everysec* for log-periodic, and *AOF-Always* for log-immediate.

Measuring data structure overheads. First of all, there is no detailed documentation on internal data structures for Redis and Memcachd. In order to conduct in-depth measurement on overheads of different in-memory data structures and their impact on workload performance, we go through the source code of both Redis and Memcached in order to identify and analyse the detailed layout and implementation of different internal data structures. This allows us to learn about the overhead of each record allocation (recall Figure 2.2 and Figure 2.3 as examples) and analyze different data structure design and overheads measured. In addition, we also measure the space overhead, the cache overhead, the overhead of read and write operations on each of the data structures, and the fragmentation overhead of different data structures. **Space overhead** is measured by storing 1,000,000, 2,000,000 and 3,000,000 records, each of 1KB in size, into different data structures for each of the four data structure container types: *list*, *set*, *sortedset* and *hash*. **Caching overhead** is recorded by tracing L1 and Last Level Cache (LLC) miss rate. Since iteration operations can best reflect the degree of CPU cache line locality, we load 10,000 records of 1KB each into the containers and measure the benchmark when executing iteration command of each container. We iterate 10,000 records randomly chosen by *GIT_KVWR*. Iteration commands include: *lrange* for list, *smembers* for set, *zrange* for sortedset, and *hgetall* for hash. **Read/write operation overhead** is measured for both small datasets and large datasets, since the performance of some data structures and their read and write operations are more sensitive to the dataset size. **Fragmentation of each data structure** is measured using *Memtier_benchmark*, *ltrace* [81] and *Valgrind* [82] using the datasets with 1,000,000 and 2,000,000 records of 1KB each. Since Redis and Memcached use different memory allocators, we test both glibc and jemalloc to gain deeper understanding of the causes for

fragmentation by recording and analyzing the memory allocation statistics.

2.3.3 Evaluation Metrics

We collect six system-level and three application-level characteristics from workloads using *SYSSTAT*, *ltrace*, *Valgrind*, *Perf* and Redis internal benchmarking commands. These characteristics collectively reveal the underlying causality details of various performance overheads, enabling an in-depth understanding of different performance overheads inherent in the in-memory key-value systems. Concretely, the following different workload characteristics are collected under different workloads, different persistency models, and different data structures:

CPU utilization (%): Three metrics are used to measure CPU utilization in user space, kernel space or storage I/O are *%user*, *%system* and *%iowait*, which show the CPU overhead occurred for performing tasks in user space, kernel space or storage I/O.

Memory and Swap usage (KB): These metrics are measuring the dynamics of memory usage and swap events per second.

Swap-in and Swap-out (page/sec): These metrics capture the number of page swap-out events and swap-in events per second, reflecting the overhead of swapping and virtual memory management during memory shortage.

L1 and Last Level Cache miss rate (%): These metrics refer to the miss rate of L1 and the miss rate of LLC cache during read operation execution for different data structures, showing the efficiency of cache line locality.

Memory fragmentation rate: this metric reflects the degree of memory fragmentation, which is calculated as the amount of memory currently in use divided by the physical memory actually used (the RSS value).

Heap allocation (byte): we measure both the total amount of memory and the total number of blocks allocated by application in heap, showing the efficiency of application memory allocation policy.

Table 2.3: Overhead of Redis during Bulk Insertion

	Overhead	Throughput	CPU *%user*	CPU *%system*	CPU *%iowait*	Memory/Swap changing (KB/sec)	Swap-in (page/sec)	Swap-out (page/sec)
non-swapping (5GB)	nosave	50206	7.39	21.43	0.66	54522	n/a	n/a
	dict expansion	46597	7.49	21.52	0.67	58558	n/a	n/a
	snapshot	49703	32.17	23.4	3.79	168753	n/a	n/a
	AOF-No	41670	9.88	21.95	2.68	87489	n/a	n/a
	AOF-No + Rewriting	25419	21.61	24.18	17.27	187148	n/a	n/a
	AOF-Everysec	41649	9.545	21.51	3.57	88896	n/a	n/a
	AOF-Everysec + Rewriting	28018	21.89	25.08	18.27	197104	n/a	n/a
swapping (10GB)	nosave	46937	10.48	22.7	3.4	50028	245	12939
	dict expansion	42335	11	22.86	11.05	46701	171	12646
	snapshot	45272	26.04	24.19	23.12	63241	4758	19471
	AOF-No	40842	9.62	21.64	10	44654	221	12380
	AOF-No + Rewriting	19465	16.43	24.81	28.97	50146	2375	13178
	AOF-Everysec	40813	10.02	20.26	7.1	44876	259	12397
	AOF-Everysec + Rewriting	19460	16.48	24.73	28.7	50972	2401	13642

Table 2.4: Overhead of Memcached during Bulk Insertion

	Overhead	Throughput	CPU *%user*	CPU *%system*	CPU *%iowait*	Memory/Swap changing (KB/sec)	Swap-in (page/sec)	Swap-out (page/sec)
non-swapping (5GB)	baseline	50611	10.45	20.69	1.51	59337	n/a	n/a
	dict expansion	50461	15.16	20.92	1.62	60630	n/a	n/a
swapping (10GB)	baseline	34250	12.25	20.05	7.61	32978	203	11990
	dict expansion	13922	6.66	8.61	36.36	13922	2435	13685

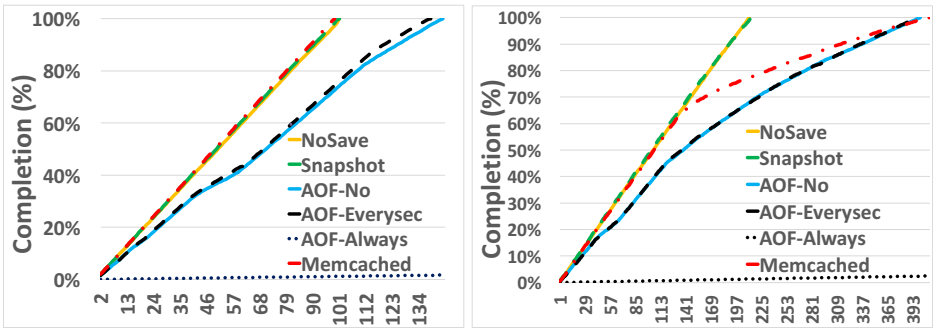
Throughput (ops/sec): this metric shows the number of operations per second for a given workload, reflecting the efficiency of request processing.

Data structure overhead (byte): we measure the maximum and the minimum space overhead of entry insertion, and these two metrics clearly reflect the space overhead of different data structures.

Operation execution time (μsec): this metric captures the CPU time of baseline read/write operation execution for a given data structure under a given workload and dataset, allowing the performance comparison for different data structures, different datasets and workloads.

2.4 Experimental Results and Analysis

In this section, we provide our measurement results and analysis for four types of performance overheads, aiming at gaining an in-depth understanding of how in-memory key-value systems handle large datasets, persistency support, virtual memory management and different in-memory data structures and how a design choice may impact on the system performance.



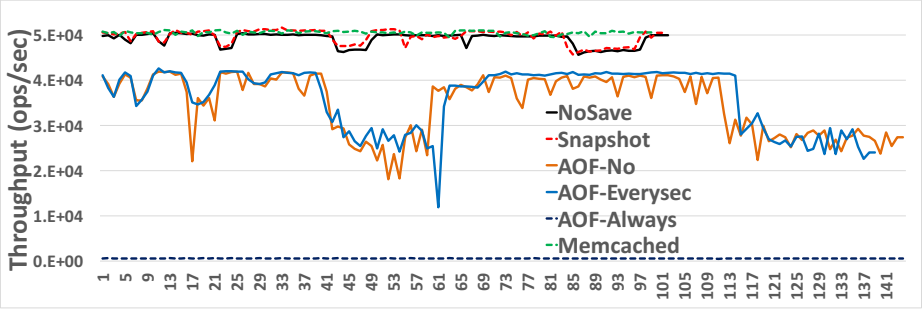
(a) Light Load (b) Heavy Load

Figure 2.11: The Completion Percent of Load with Different Overhead.

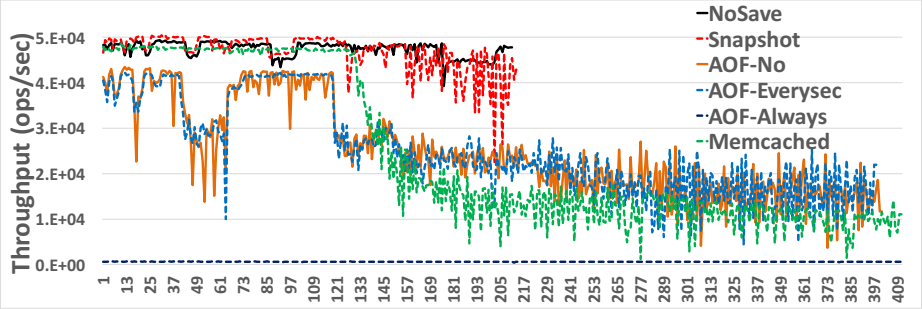
2.4.1 Overhead for Bulk Insertion of Large Datasets

In this set of experiments, we present the measurement results for dictionary expansion overhead for two scenarios: (i) bulk insertion of dataset within the allocated memory capacity and (ii) bulk insertion of dataset exceeding the allocated memory capacity. One internal threshold is set in Redis and Memcached for controlling when dictionary expansion is necessary. Once the threshold is met or exceeded, the system will start creating a double-size hash bucket and gradually moving data from existing bucket to the new one. To understand the impact of the dictionary expansion overhead on the throughput of Redis workloads, we measure the throughput of both 5GB and 10GB workloads. Figure 2.12a shows five dents, showing that the 5GB workload experienced 5 times of dictionary expansion. Table 2.3 shows that the average throughput with dictionary expansion is 46597 ops/sec and the average throughput without dictionary expansion is 50206. Thus the throughput degradation due to dictionary expansion is about 7.19% (i.e., 50206-46597)/50206=0.0719).

Another interesting observation from Figure 2.12a is that the Memcachd throughput is more stable for the same workload, even with dictionary expansion. This is because Redis is using single thread for request serving, and this single thread has to deal with both data loading and dictionary expansion. However, Memcached uses multi-threading for request serving, and dedicates one exclusive expanding thread, always ready to perform dictionary expansion task, reducing the overhead significantly. Although they both use



(a) Light Load



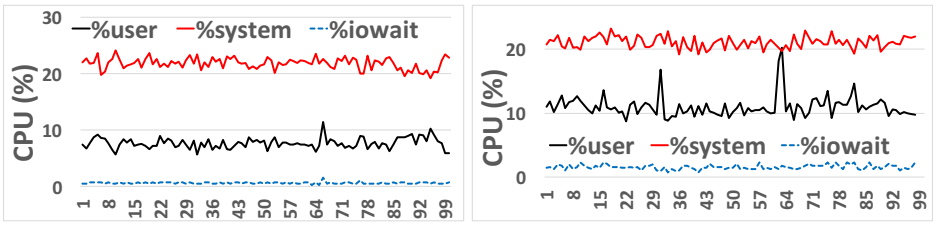
(b) Heavy Load

Figure 2.12: The Throughput of Load with Different Overhead.

the similar rehashing strategy – incremental rehashing, Memcached with multi-threading solution clearly outperforms Redis with single-thread solution. Table 2.4 presents the average throughput detail for Memcachd under 5GB (non-swapping) and 10GB (swapping) workloads.

Figure 2.13a and Figure 2.13b zoom into the dictionary expansion occurence during the elapse time from 44^{th} to 49^{th} second in Figure 2.12a for both Redis and Memcachd respectively. We observe that the CPU consumption for the dictionary expansion is negligible, and Table 2.3 gives the detailed CPU utilization of 0.66 and 0.67 without and with dictionary expansion respectively. However, the CPU utilization of Memcached in Figure 2.13b is clearly less stable compared to Redis in Figure 2.13a, and the several spikes for CPU *%user* level metric shows that the dictionary expansion thread of Memcached is busy working on the task by consuming more CPU through parallel threads.

From Table 2.3 and Table 2.4, we observe that compared to non-dictionary expansion scenario, a slight increase in memory consumption occurs during dictionary expansion,



(a) Redis CPU Utilization (b) Memcached CPU Utilization

Figure 2.13: The Overhead of Dictionary Expansion.

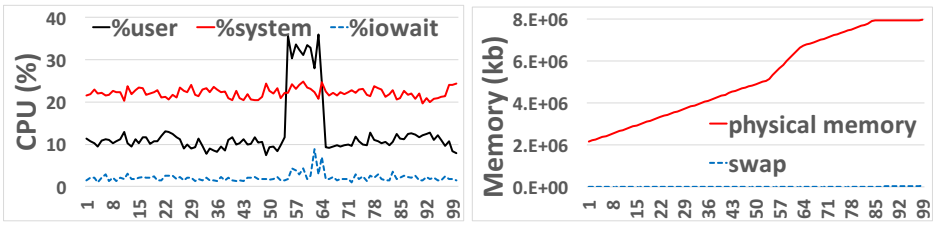
allocating 6.8% more memory per second for Redis (i.e., (58558-54522)/58558=0.068) and 2.1% more memory for Memcached (i.e., (60630-59337)/60630=0.021) respectively. Theoretically, both Redis and Memcached need to create another double-sized hash bucket for dictionary expansion, which consumes additional memory space. Experimentally, we see that such memory consumption overhead is very small, but the throughput of Redis is decreased during dictionary expansion, as shown in Figure 2.12a, leading to allocating even less memory.

Figure 2.12b shows that with 10GB workloads, the allocated memory of 8GB can no longer host all the data in memory. When persistency models are turned on, the throughput performance of all persistency models is decreasing sharply at the elapse time of 125th sec or so, compared to no persistency support scenario (NoSave). We will analyze the impact of different persistency models on workload performance in the next subsection.

2.4.2 Overheads of Persistency Models

Snapshot Overhead

Redis performs *snapshot* from elapse time 55th to 63rd second as shown in Figure 2.12a. The average throughput overhead is about 1%. The reason for such a small and negligible overhead is that Redis forks a child process by using copy-on-write for the background save purpose, thus loading data and snapshotting can run concurrently. As a result, the performance of snapshot is surprisingly good in Redis. To further understand the overhead of snapshot, we zoom in the period from 55th to 63rd second in Figure 2.14a, observing



(a) CPU Utilization (b) Memory Consumption

Figure 2.14: The Overhead of *Snapshot*.

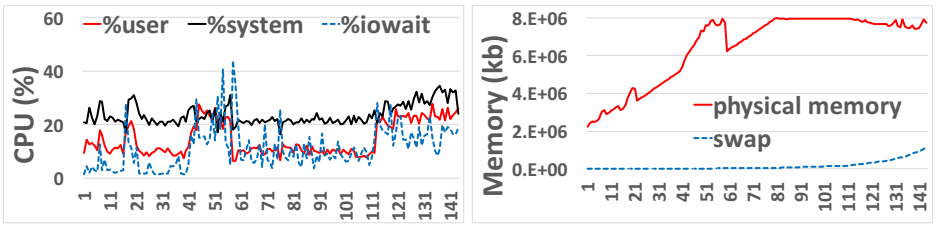
that snapshotting generates spikes for both CPU *%user* and CPU *%iowait*. The spike of CPU consumption to over 30% in user space shows the changes caused by the background RDB (*snapshot*) save process, which is forked from Redis main process, responsible for writing log files to disk and running in user space. The small spike in CPU *%iowait* means that the writing of snapshot files to disk (SSD in our test platform) are causing some I/O waits compared to the time period with no snapshotting. Another interesting observation is the sudden increase of memory consumption at 55th second as shown in Figure 2.14b, which is also caused by the background RDB save process. Although Redis adopts copy-on-write for eliminating additional memory consumption, the memory change rate is still significant with 209.51% increase (i.e., (168753-54522)/54522=2.0951). Redis creates a *rioFileIO* object, a simple stream-oriented I/O, for RDB file operating. Since buffered I/O is adopted, such as *fwrite()*, only upon the invocation of *fflush()*, any unwritten data in the stream pointed to by stream is flushed to the kernel buffer, otherwise, these data are kept in the user space buffer and occupy memory space. Moreover, data stored in kernel buffer also occupies memory space, because for performance consideration, kernel always simply copies the data into kernel buffer and gathers up all the dirty buffers, which are buffers that contain data newer than what is on disk, sorts them and writes them out to disk. Thus, the more I/O, the more memory buffer space will be used until the I/O operation of snapshotting is completed.

AOF-No Overhead

AOF-No means never *fsync*, and instead relying on the Operating system for flushing log files. We also refer to this model as log-deferred model. Compared to *nosave* mode (no persistency support), recall Figure 2.12a and Table 2.3, we observe that *AOF-No* incurs 17% throughput degradation (i.e., (50206-41670)/50206=0.17). The *AOF* file gets bigger as more write operations are performed, so Redis provides *AOF Rewrite* to rebuild the *AOF* in the background without interrupting its client services. Whenever *BGREWRITEAOF* is issued or *AOF* file is big enough, Redis will generate the shortest sequence of commands needed to rebuild the current dataset in memory. We call this *AOF Rewrite*, which happens from 17^{th} to 20^{th} second and from 41^{st} to 58^{th} second in Figure 2.12a, showing a nearly 50% throughput degradation using statistics collected in Table 2.3. Since during *AOF Rewrite* Redis rewrites all commands into a new temp file, while new commands are stored in *server.aof_rewrite_buf_blocks*. The temp file will finally replace the original *AOF* file and data stored in *server.aof_rewrite_buf_blocks* will open the new *AOF* file as append mode and insert into the end of it. All these tasks create more computation and I/O cost, resulting in significant performance degradation. *AOF Rewrite* combined with dictionary expansion can make the average throughput even worse as shown during the elapse time duration from 41^{st} to 58^{th} second in Figure 2.12a. This is also a good example to show that multiple overheads combined together could lead to seriously worse performance result.

Figure 2.15a and Figure 2.15b further zoom into the overhead of *AOF-No*. We observe that *AOF-No* is not CPU intensive but rather an I/O intensive task. This is because, according to Table 2.3, its CPU *%user* increases by 33.69%; CPU *%system* increases by 2.43%; and CPU *%iowait* increases by 306.06%. This is because additional operations for *AOF* writes (writing *server.aof_buf data* into disk file) leads to more user space computation. Whenever these writes flushed to disk by operating system, it results in large CPU *%iowait*.

We can also observe from Table 2.3, when *AOF-No* and *AOF rewrite* is combined,



(a) CPU Utilization (b) Memory Consumption

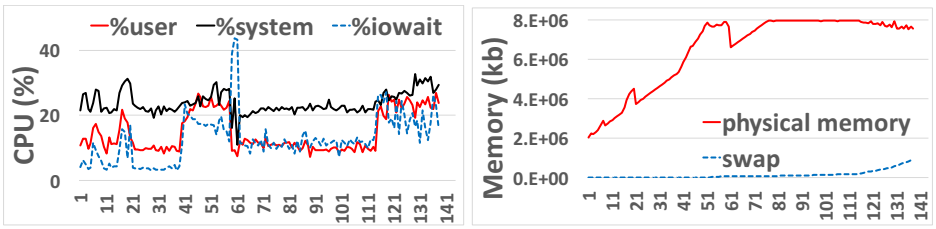
Figure 2.15: The Overhead of *AOF-No*.

it will generate a huge increase on CPU metrics, especially for CPU *%iowait*. Redis forks a separated process for completing *AOF Rewrite* task, creating more computation in user space, thus CPU *%user* increases by 192.42% (i.e., (21.61-7.39)/7.39=1.924). *AOF Rewrite* writes data into a temp file independent from *AOF* file, replace the original *AOF* file, and appends data stored in *server.aof_rewrite_buf_blocks* to the new *AOF* file, therefore, these additional I/O operations lead to higher CPU *%iowait*, which increases 2516.67% (i.e., (17.27-0.66)/0.66=25.1667).

Now we examine the memory consumption of *AOF-No*. From Table 2.3, *AOF-No* allocates 60.47% more memory space than *nosave* mode (i.e., (87489-54522)/54522=0.6047), which are needed for storing all executed but not yet persisted commands in *server.aof_buf*. When *AOF-No* is combined with *AOF rewrite*, it allocates 243.25% more memory space than *nosave* mode (i.e., (187148-54522)/54522=2.4325), because the mass of I/O operations created by *AOF rewrite* occupies huge amount of memory buffer and cache space. Figure 2.15b shows the memory consumption changes during the following elapse time durations: 2^{nd}-3^{rd} second, 7^{th}-8^{th} second, 17^{th}-20^{th} second, 41^{st}-58^{th} second, and 112^{th}-141^{st} second.

AOF-Everysec Overhead

The overhead *AOF-Everysec* is very similar to that of *AOF-No*, which makes sense since they both uses almost same *AOF* code except *AOF-Everysec* executing *fsync()* every second. By regular *sync()* execution, *AOF-Everysec* shows relatively steady throughput, avoid-



(a) CPU Utilization (b) Memory Consumption

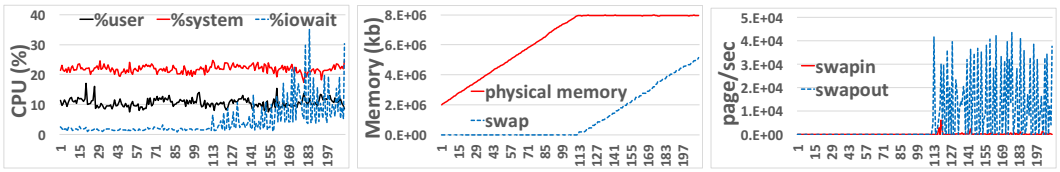
Figure 2.16: The Overhead of *AOF-Everysec*.

ing flush huge amount of data into disk at a time. The throughput overhead of *AOF-Everysec* is about 17%; the throughput overhead of *AOF-Everysec+AOF rewrite* is about 44.19%. What happened at 61st second in Figure 2.12a is that several tasks, *AOF Rewrite*, *dictionary expansion*, combining *server.aof_rewrite_buf* data into *AOF* file, deleting and renaming original *AOF* file, occur in a very short time leading to serious system performance degradation. Another potential factor is the shortage of physical memory, which we'll see shortly.

The CPU utilization and memory consumption result of *AOF-Everysec*, shown in Figure 2.16a and 2.16b, are also similar to *AOF-No*. The detailed statistics are listed in Table 2.3. We observe that the physical memory is full at 61st second, which is the root cause of throughput degradation and CPU *%iowait* spike. Operating system has to swap memory to disk to provid enough memory space for completing the running tasks. This shows that the shortage of physical memory can greatly impact the performance of Redis with *AOF* persistent model.

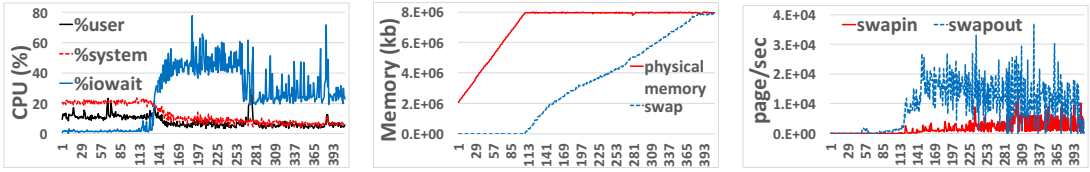
2.4.3 Swapping Overhead

We now examine the degree to which swapping may impact on the performance of dictionary expansion, and the performance of snapshot or AOF log based persistency models. Figure 2.12b shows the workloads of 10GB dataset under the available 8GB memory allocation, and thus experiencing swapping during the bulk insertion process. Figure 2.17 to Figure 2.21 show the detailed measurement results on CPU and memory utilization and



(a) CPU Utilization (b) Memory Consumption (c) Swapping

Figure 2.17: The Overhead of Redis *Dictionary Expansion* with Swapping.

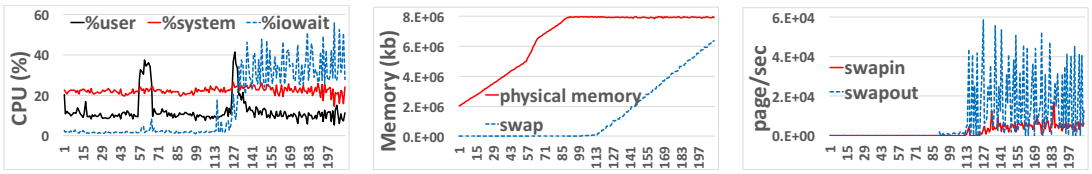


(a) CPU Utilization (b) Memory Consumption (c) Swapping

Figure 2.18: The Overhead of Memcached *Dictionary Expansion* with Swapping.

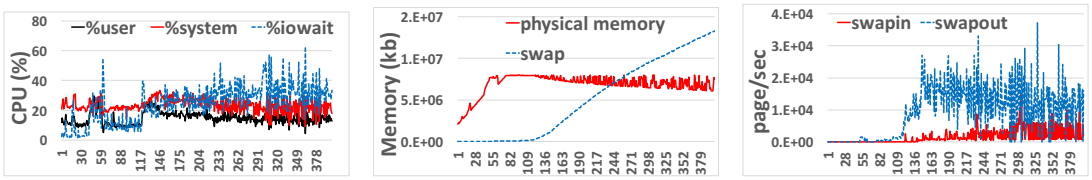
swapping activities for dictionary expansion scenarios, the snapshot and AOF log based persistency models using the 10GB dataset.

For ***dictionary expansion***, the throughput degradation of Redis is about 9.15% compared with non-swapping scenario according to Table 2.3. From Figure 2.12b, the degradation occurs from 177th to 202nd second. More interestingly, the impact of swapping on Memcached is even bigger, and the throughput degradation of Memcached is about 72.41%, compared with non-swapping scenario, which occurs from 135th till the end as shown in Figure 2.12b. The throughput of Memcached decreases to about 13,922 ops/sec by Table 2.3, which is lower than the lowest throughput for the Redis *AOF* model. From Figure 2.17c and Figure 2.18c, we make another interesting observation: after the 111th second, due to the shortage of physical memory, operating system starts to swap-out some data into disk in order to provide enough memory space for loading in new data. Bulking loading and *dictionary expansion* fiercely compete memory resource, thus some swap-out data probably need to be swapped in by *dictionary expansion* task. Both Figure 2.17c and Figure 2.18c show that both Redis and Memcached have to swap-in pages. Because of different memory management policy adopted, Memcached shows a lot more swap-in pages than Redis, this illustrates the reason for sharp performance reduction of Memcachd shown



(a) CPU Utilization (b) Memory Consumption (c) Swapping

Figure 2.19: The Overhead of *snapshot* with Swapping.

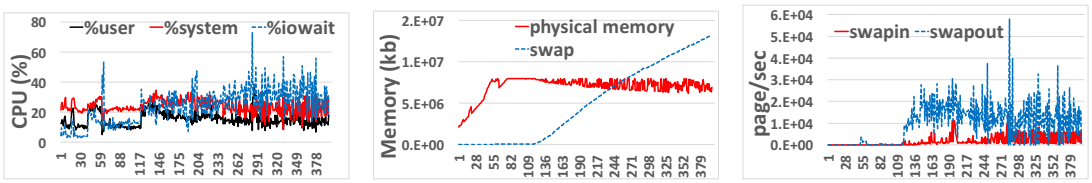


(a) CPU Utilization (b) Memory Consumption (c) Swapping

Figure 2.20: The Overhead of *AOF-No* with Swapping.

in Figure 2.12b.

For ***snapshot***, the throughput of *snapshot* with swapping is 45272 and the throughput of *nosave* with swapping is 46937. Combining with experiment under non-swapping condition, we conclude that snapshot consumes very little system resource, and generates almost same throughput as *nosave* mode. However, since *dictionary expansion* seriously impacts the performance of in-memory key-value systems in both non-swapping and swapping condition, snapshotting while expanding dictionary makes the system slow down, as shown in Figure 2.12b and Figure 2.19, starting at the elapse time of 177th second. During snapshotting, CPU *%iowait* increases to about 23.12%, which is more than 600% of non-swapping scenario. CPU *%system* slightly increases, since operating system is busy working on I/O system calls. Due to I/O wait, CPU *%user* shows a decrease compared with non-swapping scenario. In Figure 2.19b, the slope of memory changing with swapping is 37.48% of non-swapping case. This is partly caused by the slow swap allocation and the huge swap-out operations, as shown in Figure 2.19c. Snapshot process needs to iterate every key-value pair of each database. When these data were swap-out to disk, the operating system has to swap them back to memory again for taking snapshot. Moreover, since *snapshot* process writes every data of Redis into RDB file, operating system also has to provide



(a) CPU Utilization (b) Memory Consumption (c) Swapping

Figure 2.21: The Overhead of *AOF-Everysec* with swapping.

enough memory buffer in both user space and kernel space, thus, according to Table 2.3, *snapshot* shows more swap-out and swap-in events compared to *nosave* and some of the *AOF* models with swapping condition.

For ***AOF-No*** and ***AOF-Everysec***, similar throughput can be observed. By Table 2.3, the throughput overhead of *AOF-No*, *AOF-Everysec*, *AOF-No+rewrite* and *AOF-Everysec+rewrite* are 12.99%, 13.05%, 58.53%, and 58.54% respectively, compared to *nosave* with swapping. From Figure 2.12b, Figure 2.20 and Figure 2.21, we observe that *AOF rewrite* process is very memory consuming and the physical memory is full at the 80th second. Also the throughput becomes unstable from 80th to 119th second. After 119th seconds, *AOF rewrite* starts running till the end, the memory is totally full at the 119th second, and *AOF rewrite* requires huge amount of memory that OS cannot provide immediately. Thus, Figure 2.12b shows a sharp decrease of throughput at 119th second. Moreover, since *AOF rewrite* job includes large amount of read and write operations, operating system has to swap-in and swap-out constantly as shown in Figure 2.20c and Figure 2.21c, leading to memory thrashing, which seriously worsens the system performance.

The insight we gained from this set of experiments is four folds. First, adding one additional thread dedicated to dictionary expansion could improve throughput for in-memory key-value systems like Redis. Second, *AOF-Rewrite* is a CPU and memory intensive task, which results in obvious degradation of system performance. If swapping also occurs, it can generate severely worse throughput. Third, swapping seriously impacts on the performance of Memcached. Finally, *snapshot* based persistency model in comparison shows the best throughput performance compared with other persistent models for both non-swapping

Table 2.5: Overhead of Different Data Structure - 1

Container	Data Structure	Space Size(1,000,000 records × 1KB dataset)			Cache	
		Memory Size (byte)	RDB Size (byte)	AOF Size (byte)	L1-dcache Miss Rate	LLC Miss Rate
List	ziplist	1,368,895,696	1,052,745,971	1,396,000,023	7.15%	9.48%
	linkedlist	4,136,895,488	1,070,888,920	1,396,000,023	8.30%	10.42%
Set	intset	600,894,208	477,888,736	1,444,999,448	7.05%	7.94%
	hashtable	5,122,687,312	1,084,888,464	1,444,999,423	6.73%	10.86%
SortedSet	ziplist	2,229,839,464	904,183,753	2,553,617,665	7.92%	9.33%
	skiplist	8,051,753,632	1,928,523,295	2,553,611,249	8.36%	11.14%
Hash	ziplist	1,880,895,488	1,266,746,967	2,037,000,023	6.94%	10.12%
	hashtable	6,552,895,488	1,452,401,409	2,037,000,023	7.09%	14.57%

Table 2.6: Overhead of Different Data Structure - 2

Container	Data Structure	Space Overhead		Fragmentation	
		Min Overhead (byte)	Max Overhead (byte)	Jemalloc	Glibc
List	ziplist	75+m+ (2+n)*g	75+m+ (10+n)*g	1.01	1.05
	linkedlist	112+(48+n)*g+m	112+(48+n)*g+m	1.02	1.27
Set	intset	72+m+2*g	72+m+2*g	1.03	1.1
	hashtable	192+(48+n)*g+m	224+(48+n)*g+m	1.03	1.25
SortedSet	ziplist	75+(5+n)*g+m	75+(21+n)*g+m	1.02	1.03
	skiplist	264+(72+n)*g+m	296+(72+n)*g+m	1.02	1.25
Hash	ziplist	75+m+ (4+n+k)*g	75+m+ (20+n+k)*g	1.03	1.03
	hashtable	192+(72+k+n)*g+m	224+(72+k+n)*g+m	1.03	1.22

and swapping workloads.

2.4.4 Overhead of Different Data Structures

We examine four types of overheads for different data structures: memory space overhead, cache overhead, read/write operation overhead and fragmentation.

Space Overhead

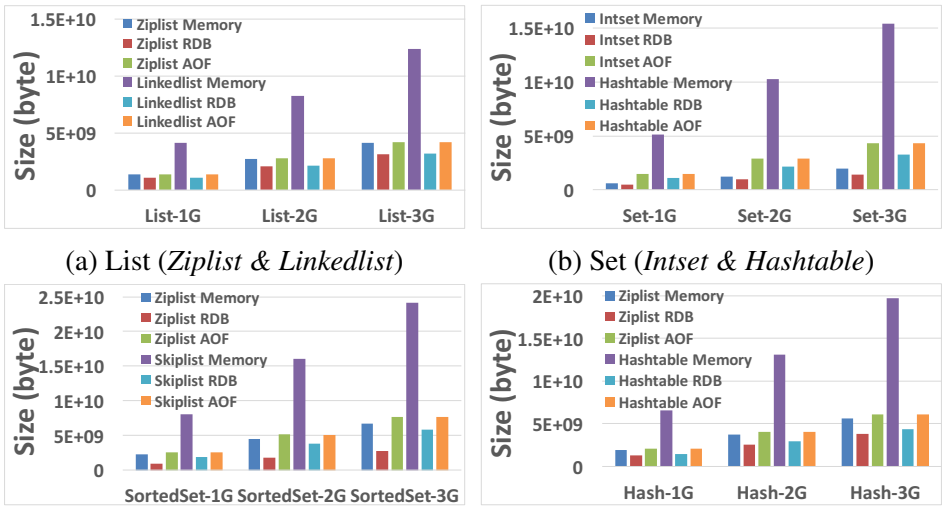
For the same dataset, the memory space allocated by *linkedlist* is 3.02 times of the size of *ziplist*. By using the compact internal structure, *ziplist* saves lots of physical memory. Although *linkedlist* is more flexible for insertion and deletion operations, it uses several additional data structures. For example, each entry has to allocate at least additional 83 bytes for list data structure. This space overhead is serious when storing lots of small records.

Given that *AOF* file stores execution commands, the *AOF* file size for different data structures are the same as long as they are using the same command to store data. In contrast, *RDB* file is a compressed binary file, which stores database information, key-value pair data and encoding information. The size of *RDB* file depends on which data structure we use. By using a compressed log file, it saves considerable disk space, even if a memory consuming data structure is used. For example, Figure 2.22a shows that the size of *linkedlist RDB* file is only 25.87% of its memory allocation. These design principles also apply to *set*, *sortedset* and *hash*.

For *intset* and *hashtable* comparison, we only store integer values, as this will maximize the space overhead difference between *intset* and *hashtable*. *Intset* memory allocation is only 11.73% of *hashtable*. Since *intset* uses integer array to store all values, extremely reducing the memory size. Thus, *intset* is one ideal container for keeping pure integer data. Conversely, *hashtable* doesn't support integer storage, instead everything is stored as *char*, thus the space overhead is even bigger, especially for long numbers, because every digit occupies one byte in *hashtable*, but in *intset*, whole number merely occupies two bytes. *Intset AOF* file is even 2.4 times bigger than its memory allocation (Figure 2.22b), from another perspective this proves that using integer array can efficiently save memory space.

Data Structure of *skiplist* (recall Figure 2.7) shows its complexity. It can be viewed as a combination of *hashtable* and *skiplist* to enhance both search and sort performance. However, the drawback is obvious on its high memory consumption. It occupies 7.50GB for 1GB raw dataset (see Figure 2.22c), which is highest memory allocation among all data structures supported by Redis.

In comparison, *ziplist* is a good choice, since for 1GB dataset, it only occupies 2.08GB memory. *Ziplist* can be used as *list*, *sortedset* and *hash*. For *hash*, *ziplist* is used in a way similar to *sortedset*. The only different is that it uses the position of storing score to store keys. Thus, key-value pairs are stored in continuous memory space, efficiently reducing the memory consumption. However, the downside is the lower search performance. For



(a) List (*Ziplist & Linkedlist*) (b) Set (*Intset & Hashtable*)

(c) Sortedset (*Ziplist & Skiplist*) (d) Hash (*Ziplist & Hashtable*)

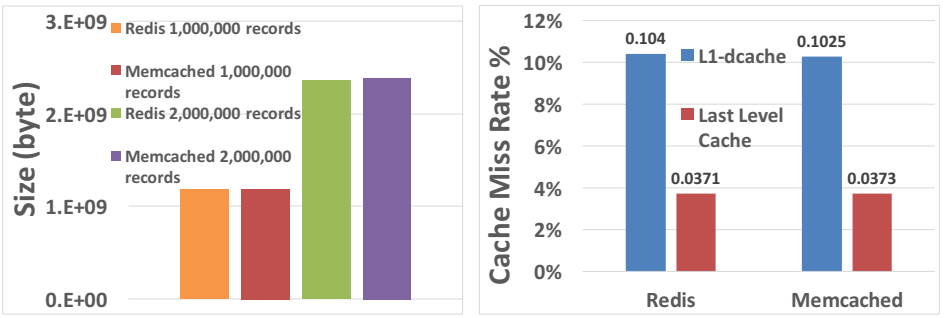
Figure 2.22: Memory and Storage Consumption of Different Data Structures.

hashtable, although it has great search performance, it consumes too much space. For 1GB raw dataset, it occupies about 6.10GB memory (Figure 2.22d), 6 times of the raw dataset size.

Given that Memcached only supports simple key-value pairs, we can only compare space overhead between Redis and Memcached by storing dataset of simple key-value pairs. Figure 2.23a shows that Memcached uses 1.26% and 1.30% more memory than Redis for loading 1,000,000 and 2,000,000 dataset records generated by *memtier_benchmark*, each record is 1KB.

Caching Overhead

Caching locality is one of key factors when designing system data structure since main memory reference is 200× lower than L1 cache reference. Array like data structures, *ziplist* and *intset*, are contiguous memory blocks, so large chunks of them will be loaded into the cache upon first access. This makes it comparatively quick to access future elements of the array. *linkedlist*, *hashtable*, and *skiplist* on the other hand aren't necessarily in contiguous blocks of memory, and could lead to more cache misses, which increases the time it takes to access them.



(a) Space overhead (b) Caching overhead

Figure 2.23: Space and Cache Comparison Between Redis and Memcached.

We measured *L1-dcache* and *LLC* miss rate of Redis, as shown in Table 2.6. *ziplist* beats all competitors, *linkedlist* for list, *skiplist* for sortedset, and *hashtable* for hash. Since the huge latency between L1 cache reference and main memory reference, although *ziplist* is slow on searching, better cache locality design could compensate it in some degree. Thus what you lost in terms of scanning the aggregates containing N elements, you win back because of better locality. This advantage could be magnified for small datasets.

Intset is another data structure better using cache locality. Even though it greatly reduces the memory allocation by using integer array, this using coincidently could also benefit for its performance. In our experiment, the L1 data cache miss rate of *intset* is similar as *hashtable*, but the Last Level Cache miss rate is 26.89% smaller. The result shows that array structure does better use CPU cache locality. But this advantage only works well for iteration operations, since *intset* is implemented as a sorted integer array, *add/remove/search* operations have to operate data in a binary search way, reducing the benefits of CPU cache locality.

We also compared cache miss rate of both Redis and Memcached. For the same reason, we only measured the simple key-value pairs dataset. As shown in Figure 2.23b, they generate very similar miss rate of both L1 and LLC.

Table 2.7: Operation Performance (all results are average CPU time (usec) consumed by commands)

	Dataset-1 (each record = 100 × 10byte)					Dataset-2 (each record = 1000 × 10byte)				
List Data Structure	rpush	rpop	lpush	lpop	lrange	rpush	rpop	lpush	lpop	lrange
ziplist	0.8	0.84	1.01	1.07	5.04	0.93	1.01	2.91	2.92	45.5
linkedlist	0.63	1	0.65	0.98	4.84	0.69	1.21	0.69	1.12	38.44
Set Data Structure	sadd	scard	smembers	sismember	srem	sadd	scard	smembers	sismember	srem
intset	1.01	0.52	13.98	0.73	0.85	1.82	0.66	114.61	1.02	1.59
hashtable	1.28	0.56	28.33	1.23	1.11	1.12	0.69	238.21	1.1	0.97
SortedSet Data Structure	zadd	zrem	zrange	zrank	zscore	zadd	zrem	zrange	zrank	zscore
ziplist	3.11	2.63	5.91	1.91	3.74	14.36	15.67	55.08	10.92	12.34
skiplist	1.73	1.64	9.92	1.95	2.71	2.02	2.41	84.98	2.88	2.9
Hash Data Structure	hset	hget	hgetall	hdel	hexists	hset	hget	hgetall	hdel	hexists
ziplist	2.33	1.49	10.29	1.94	1.99	15.33	7.13	152.89	11.19	14.95
hashtable	1.67	1.32	15.25	1.42	1.16	1.19	1.01	204.36	1.32	1.18

Read/Write Operation Performance

Different data structures have their own pros and cons. Table 2.7 compares various data structures with read/write operations. For list data structures, we mesured the performance of push/pop operation on both left-side and right-side, since *ziplist* shows maximum 3 times performance variation on different side operation, which is caused by the internal design of *ziplist*. For set data structures, we found that performance of *intset* operations is sensitive to the size of dataset, unlike *hashtalbe*. For sortedset data structures, we tested the add, remove, check, and iterate operations, showing that *skiplist* outperfoms *ziplist* for all operations except iterate operation, since the continuous memory structure makes *ziplist* better suited for scanning records. For hash data structures, we benchmarked set, get, remove, and iterate operations, showing that *hashtalbe* always keeps excellent performance for expanding dictionary.

Concretely, as shown in Table 2.7, for the list operation performance experiment, *push* operation of *linkedlist* is always faster than *ziplist*. Since every *push* and *pop* operation requires a reallocation of the memory used by the *ziplist*, the actual complexity is related to the amount of memory used by the *ziplist*. However, *linkedlist push* and *pop* operation simply add or remove *listnode* and update pointers, leading to faster and constant speed. This also explains why *ziplist push* and *pop* operation is slower in the Dataset-2 experiment where each record is 1000× 10 bytes instead of 100× 10 bytes in the Dataset-1 experiment, while *linkedlist* shows steady performance as the changing of dataset. *Ziplist lpush*

and *lpop* are obviously slower than *rpush* and *rpop*, even serious in Dataset-2 experiment. The reason is that inserting new entry at the beginning of *ziplist* needs additional memory movement by using *memmove*, providing enough space for new entry, the overhead of which depends on the length of datasets. For *lrange*-iternation operation, both *ziplist* and *linkedlist* show good and similar number. *Ziplist* uses continuous memory space to store entries. Every entry knows the length of the previous entry, by which it can iterate from the end of *ziplist* to the beginning. This structure is very similar as *linkedlist*, which is why they both show close benchmarks. In summary, *linkedlist* performs well for both experiments, the performance has nearly no connection with dataset size except *lrange* operation. Since *ziplist* needs additional time to move memory, it shows worse result as the increasing number of entries, especially for *lpush* and *lpop* operations.

In set operation performance experiment (Table 2.7), *intset* outperforms *hashtable* for every operation benchmark in Dataset-1 experiment, showing that *intset* is a good candidate for small dataset. However, in Dataset-2 experiment, we see a performance degradation of *intset*, while *hashtable* almost is insensitive to the amount of entries. The reason why *intset* is slower in the second experiment is that *intset* is implemented as a sorted integer array, whose complexity is O(log N), which means if the newly added/removed entry is not at the rear of array, *intset* has to not only resize the array, but also adjust the position of rest entries in order to guarantee the ordered structure of itself. Therefore, the more entries the more time spent for *sadd*, *srem* and *sismember* operations. For better performance and less collision of hash, Redis keeps expanding the bucket of dictionary. This experiment shows the benefit of dictionary expansion. Although more entries in Dataset-2, *hashtable* shows better *sadd/srem/sismember* benchmarks, since Redis rehashed the dictionary in order to provide nearly constant performance. The continuous memory allocation design helps *intset* beat *hashtable* for iteration (*smembers*) benchmark.

In sortedset operation performance experiment (Table 2.7), considering the structure of *ziplist*, *zadd*, *zrem*, *zrank* or *zscore* needs to iterate the whole *ziplist* O(N) until finding

the correct position to operate, which means the more entries the more runtime. *Skiplist* however can directly get score value through dictionary searching in O(1) and then calculate rank through searching *skiplist*, which is very fast compared with *ziplist*. *Skiplist* adds/removes entries by using efficient skiplist structure, which is good at sorted set. Through both experimental result, we can find that it always performs very good benchmark, even if dataset increases. Again, continuous memory allocation and good cache locality make *ziplist* outperform *skiplist* for *zrange*-iteration operation in Dataset-1 experiment, however, *ziplist* has one apparent drawback, slow locating start entry. It has to spend O(N) time to locate, comparing with *skiplist*'s O(log N) locating algorithm, this is very slow. Thus, *ziplist*'s good performance merely works for small dataset. The slowness of *zadd* and *zrem* of *ziplist* is also partially caused by reallocation of memory, the more entries the more time spent for reallocating memory and moving entries. As such, a great performance decrease is found in Dataset-2 experiment.

In Hash operation performance experiment (Table 2.7), *ziplist* is very dataset sensitive, but *hashtable* is not. *Hset*, *hget*, *hdel* and *hexists* of *ziplist* all need iterating the whole *ziplist* until finding the entry, thus these operations are very time consuming. As the number of entries increasing, *ziplist* shows performance degradation. However, *hashtable* performs steadily since it only needs to update memory space for the entry operated. Even for very large datasets, *hashtable* can also shows excellent result. Like we discussed in set operation performance experiment, by dynamically adjusting the bucket of dictionary, *hashtable* always keeps good benchmark. *Hashtable* is even faster in Dataset-2, which is 10 times bigger than Dataset-1.

Finally, we compare Redis and Memcached by using *set* and *get* operation. Figure 2.24 shows the results. Redis and Memcached show almost the same *get* benchmarks and output are quite stable. The performance difference is only 0.0059%. However, the unstable *set* throughput of Redis is caused by *dictionary expansion* and its single thread design. It is worth noting that Redis generates better performance than Memcached during the period

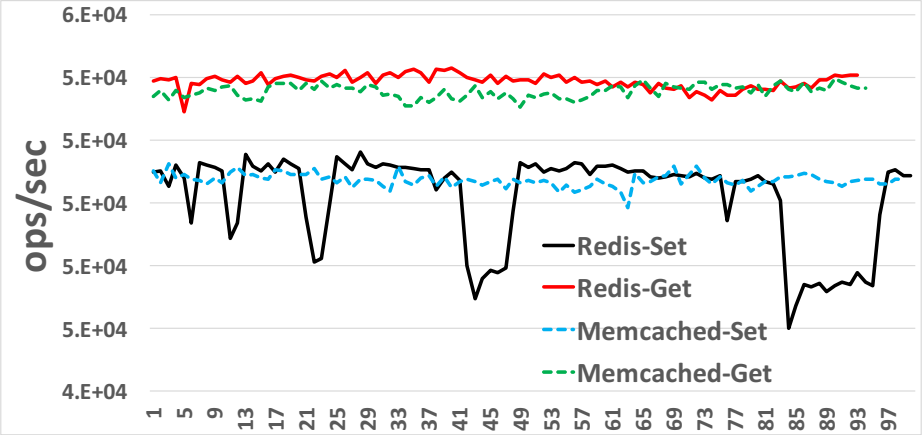


Figure 2.24: *set/get* Performance of Redis and Memcached

without *dictionary expansion*. But when dictionary expansion occurs, Memcached outperforms Redis on *set* operation by 1.08%. Clearly, optimization for dictionary expansion, such as using a dedicated thread, is critical for improving throughput performance of Redis.

Fragmentation

We now measure the degree of fragmentation for Redis and Memcached memory allocators, jemalloc and glibc. Recall Table 2.6, jemalloc shows stronger ability of solving fragmentation than glibc, nearly reaching the upper bound of the best result. The average of fragmentation for all 8 data structures is 1.02, whereas the average of fragmentation by glibc is 1.15. This is an important reason of why Redis changed its default memory allocator to jemalloc since version 2.4.

Another observation is that *ziplist* and *intset* shows better fragmentation result than *hashtable*, *skiplist*, and *linkedlist*. Contiguous-memory allocation minimizes internal fragmentation at the expense of possible external fragmentation, *ziplist* and *intset* are data structures like this. Since the physical memory in our experimental environment is big enough for dataset, the external fragmentation is not obvious. Every time adding or removing entries from *ziplist* or *intset*, Redis resizes the contiguous memory space first and then moves all impacted entries to the correct position in order to guarantee the sequence and memory continuity. However, other data structures just simply allocate and deallocate as they want,

Table 2.8: Heap Measurement - 1

key-value system	Memory Usage	Memory Wasted
Redis1,000,000 records	1,180,042,265	na
Memcached 1,000,000 records	1,195,074,650	102,131,302
Redis 2,000,000 records	2,356,863,304	na
Memcached 2,000,000 records	2,388,001,817	203,214,028

Table 2.9: Heap Measurement - 2

key-value system	Heap						
	Total Allocation (byte)	Total Block Allocated	Max Live Allocation (byte)	Max Live Block	Num of Malloc()	Num of Free()	Block Size (byte)
Redis1,000,000 records	1,271,538,266	7,013,016	1,112,739,857	4,011,659	7,001,265	3,000,691	181
Memcached 1,000,000 records	1,201,184,070	1,287	1,193,318,638	1,276	1,136	4	933320
Redis 2,000,000 records	2,547,742,990	14,013,327	2,226,148,187	8,011,918	14,002,762	6,001,225	181
Memcached 2,000,000 records	2,402,036,870	2,418	2,385,782,830	2,406	2,266	5	993398

leading to scattered data blocks. Therefore, the using of contiguous-memory leads better fragmentation result.

Next, we measure the heap allocation for comparing the fragmentation of Redis and Memcached as shown in Table 2.9.

The total allocation for Redis is slightly larger than that of Memcached, since Redis uses lots of structs representing different intermediate objects, which is relatively more complex than Memcached. However, Redis releases 7.19% (1,000,000 records test) and 7.49% (2,000,000 records test) memory during its runtime and Memcached only releases 0.58% (1,000,000 and 2,000,000 records test), thus the final memory usages of both are almost the same. As a result, Memcached uses 1.27% (1,000,000 records test) and 1.32% (2,000,000 records test) more memory than Redis.

Without using jemalloc, Memcached minimizes fragmentation by avoiding arbitrarily sized entry allocations. It predefines several dozens of size classes, called *slabclasses*. The storage is broken up into 1MB pages. Each page is then assigned into slab classes as necessary, then cut into chunks of a specific size for that slab class. When storing items, they are pushed into the slab class of the nearest fit. Comparatively, every insertion operation needs *malloc*( ), leading Redis generates 6000$\times$ more *malloc*( ) function calls than Memcached. Moreover, since Redis and Memcached both have similar memory usage and Redis allocates 5449 (1,000,000 records test) and 5795 (2,000,000 records test) times

more memory blocks than Memcached, it shows that Redis allocates lots of small memory blocks and however Memcached allocates only huge blocks. The average block size of Redis is 181 (both tests) bytes and that of Memcached is 933320 (1,000,000 records test) 993398 (2,000,000 records test) bytes, showing a totally different memory management strategy between Redis and Memcached.

By measuring memory usage results, Redis indeed occupies less memory space. In the latest version of Memcached, the chunk size growth follows exponential rate of 1.25, so the slabs size will be 96, 120, 152, and so forth. This means that the space waste is close to 8.5% by using slab allocator. Given that the unused memory in slab class is still available for Memcached to use in the future. Thus, we conclude that Memcached achieves as good fragmentation as Redis does.

From this set of experiment, we make the following important observations: (1) Array-like structure makes *ziplist* and *intset* good for memory consumption and cache locality at the expense of low performance for big dataset. (2) The operation performance is really related to specific data structure and workload. (3) The performance of right-side operations (*rpush, rpop*) of *ziplist* are always better than that of left-side operations (*lpush, lpop*). (4) Thanks to *dictionary expansion*, *hashtable* might even show better performance with bigger dataset. (5) *skiplist* is a lot faster than *ziplist* for large datasets, such gap cannot be compensated by the use of continuous memory space. (6) By using different memory management policy, jemalloc vs. slab allocator and arbitrary allocation vs. pre-allocation, both Redis and Memcached achieve good performance while reducing the fragmentation.

2.5 Related Work

We are aware of two existing experimental studies of in-memory key-value systems [83, 84], but none has examined whether and how much the overhead of data structure and internal tasks may impact on the efficiency of using memory and the performance of in-memory systems. Furthermore, [83] only considers two experimental metrics, while [84]

lacks comprehensive experimental result for all combination of data structures and system overhead. Finally, neither study considers the system that we include, Redis.

Although the study of the performance of in-memory key-value system has received only little attention, there are a number of related studies focusing on memory optimization that can shed light on the relevance of this work and its methodology. MICA [72] highlights its new data structures-circular logs, lossy concurrent hash indexes, and bulk chaining-handle both read and write intensive workloads at low overhead, since it exploits cache semantics to provide fast writes and simple memory management. By providing data structures and access methods specifically designed for storing, searching, and processing data in main memory, IBM solidDB [85] outperforms ordinary disk-based databases even when the latter have data fully cached in memory. MemC3 [86] presents optimistic hashing, compact CLOCK-based eviction algorithm and optimistic locking in order to provide higher throughput using significantly less memory and computation. SILT [87] designs and implements three basic key-value stores that uses new fast and compact indexing data structures, each of which places different emphasis on memory-efficiency and write-friendliness. By synthesizing these basic stores, SILT could balance the use of memory, storage, and computation to craft a memory-efficient, high-performance key value store. Finally, there are multiple studies offering the optimization in production - Facebook's Memcache [88] and Twitter's Twemcache [89].

2.6 Conclusion

We have presented a performance evaluation and analysis of in-memory key-value systems. To the best of our knowledge, this is the first in-depth measurement study on critical performance and design properties of in-memory key-value systems, such as the use of different internal data structures, different persistency models and different policies for memory allocators. We measure a number of typical overheads such as memory space, caching, read/write operation performance, fragmentation, and workload throughput per-

formance, to illustrate the impact of different data structures and persistency models on the throughput performance of in-memory key-value systems. We conjecture that the multiple factors on memory efficiency will provide system designers and big data users with a better understanding on how to configure and tune the in-memory key-value systems for high throughput performances under different workloads and internal data structures.

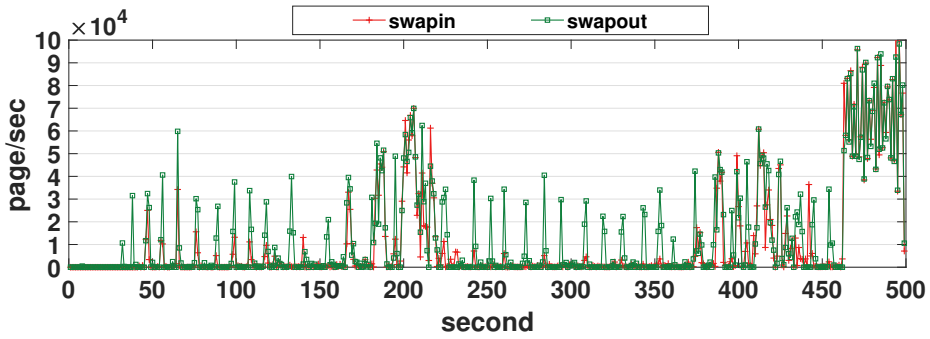
CHAPTER 3

EFFICIENT VM EXECUTION WITH FASTSWAP

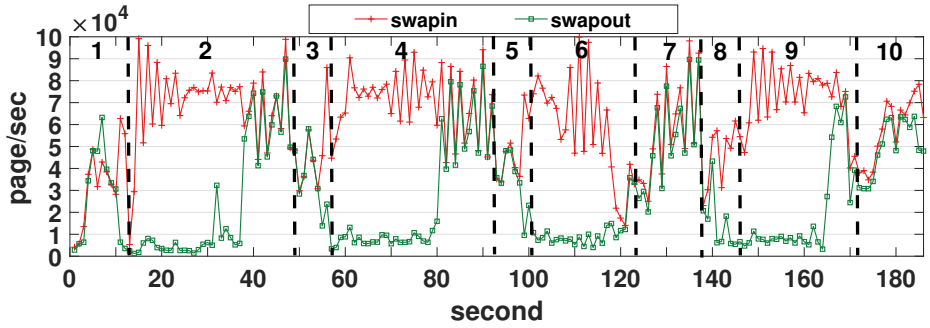
Many big data applications are memory-intensive workloads and perform iterative analytics algorithms. When the dataset used in each iteration of the analytic job exceeds the physical memory of their allocation, this type of workloads suffers from serious performance degradation or experience out of memory error. Existing proposals focus on estimating working set size for accurate resource allocation of executors, but lack of desired efficiency and transparency. This chapter presents an efficient shared-memory based memory paging service, called *FastSwap*. The design of *FastSwap* makes a number of original contributions. First, *FastSwap* improves VM memory swapping performance by leveraging idle host memory and redirecting the VM swapping traffic to the host-guest compressed shared memory swap area. Second, *FastSwap* develops a compressed swap page table as an efficient index structure to provide high utilization of shared memory swap area through supporting multi-granularity of compression factors. Third, *FastSwap* provides hybrid swap-out and proactive swap-in to further improve the performance of shared memory swapping. Finally, *FastSwap* is by design light-weighted and non-intrusive. We evaluate *FastSwap* using a set of well-known big data analytics workloads and benchmarks, such as Spark, Redis, HiBench, SparkBench and YCSB. The results show that *FastSwap* offers up to two orders of magnitude performance improvements over existing memory swapping methods and more than four orders of magnitude faster than conventional disk based VM swapping facility.

3.1 Introduction

The growth rate of big data in cyberspace continues to outpace both hardware and software technologies. Many machine learning algorithms and data analytics jobs experience tem-



(a) Loading



(b) Iteration

Figure 3.1: Logistic Regression Swapping

poral memory usage variations across virtual machines or executors on the same machine and across a cluster. In virtualized clouds, the analytics workloads enjoy optimal performance when their data can fully fit into the physical memory allocated prior to runtime. However, when the data exceeds the allocated physical memory of the analytic computation, the applications will experience serious performance degradation due to excessive memory swapping, which may lead to out of memory error, even though the host machine has enough free memory. Figure 3.1a and Figure 3.1b show periodic burst of memory swapping events during each iteration of the logic regression workloads, because the input dataset cannot fully fit into the available physical memory and some portions of the dataset are stored in the swap backing storage on the disk swap partition of the OS while keeping their memory foot-print through the conventional OS virtual memory facility.

Figure 3.1a shows the VM swapping events of the LR workload with 5 million samples during data loading and the swap-out events are the dominating swap traffic. Fig-

ure 3.1b shows the VM swapping events for the subsequent 10 iterations and the swap-in traffic dominates with relatively moderate amount of swap-out events during each iteration, showing periodic patterns of VM memory swapping traffic.

Existing Solutions. Several solutions have been proposed to address the VM memory pressure problem. The balloon driver is proposed [31, 32] as a dynamic memory balancing mechanism for efficiently moving memory from one VM to another via host with inflation and deflation operations. However, when to start ballooning and how much memory ballooning is adequate have been a challenging problem and rely on manual administration. Several existing efforts propose to estimate the working set size of each VM at run time and add or remove additional memory dynamically based on its estimated memory demands [90, 7, 8, 10]. However, recent study [91] has shown that accurate estimation of VM working set size is difficult under changing workloads. Consequently, any delay in VM memory consolidation, such as ballooning, can cause the VM under memory pressure to experience increased memory swapping events when the application cannot fit their working set fully in its allocated physical memory. Excessive memory swapping can lead to serious performance degradation for applications, which may suffer out of memory induced system crash due to high latency swapping events induced timeout.

Figure 3.2 shows some experimental results of Redis [2] (a popular main-memory key-value system). The dataset uploaded to Redis is slightly larger than the VM physical memory allocated at VM initialization time, and Redis transparently stores some data in the form of swap-out pages on external swap storage (e.g., disk in conventional OS). A Redis client is running a key-value workload of 50% uniform read and 50% uniform write generated by YCSB [77]. It is observed from Figure 3.2 that the throughput of Redis server is increasing during the first 10 seconds, because all its data access is served in memory and there is no memory swap. However, as soon as Redis has to access the portion of its dataset that are not residing in its physical memory, the Redis server experiences drastic throughput degradation due to heavy memory swapping events. At the 15th second, the balloon driver

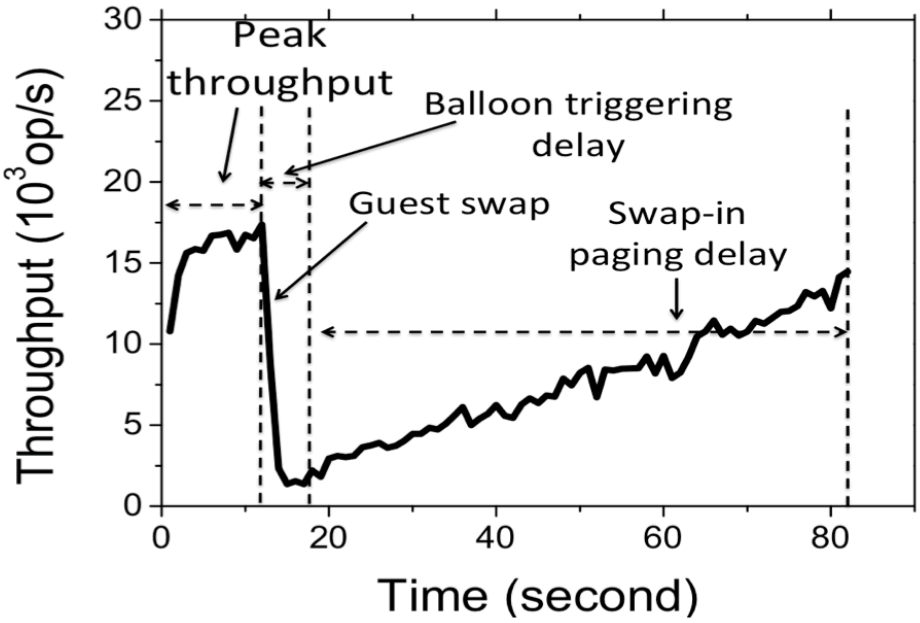


Figure 3.2: Effect of VM Swapping on Redis Performance

is turned on and it takes only a couple of seconds to balloon 4GB memory from host to VM even when the CPU utilization on both the VM and the host is at 60%-80%. This figure also shows two interesting facts: (1) The Redis server starts swap-in pages at the 18th second (when its memory is inflated via ballooning, and the Redis performance is recovering slowly. (2) Even with sufficient memory, Redis still takes more than 60 seconds to recover to its normal performance. We observe that this is mainly due to the large number of page-fault triggered swap-in operations and the series of disk I/Os to read data from disk swap area to the Redis working memory.

Contributions of the *FastSwap*. We have shown through experiments two important observations. First, big data analytic workloads often experience memory swapping events when their datasets cannot fully fit into the physical memory allocated (recall Figure 3.1a and Figure 3.1b). Second, applications (e.g., Redis) suffer from performance degradation when their working set no longer fully fit into the physical memory even when other VMs on the same host have unused idle memory (recall Figure 3.2). Even after sufficient memory is ballooned from host to the VM, the performance of application takes time to

recover to its peak performance due to slow and inefficient memory swap-in operations. In this chapter, we address these problems by presenting *FastSwap*, a host-coordinated shared memory swapper with multi-granularity compression on memory swap pages. We show that *FastSwap* can significantly improve the runtime performance of big data analytics workloads in virtualized Clouds. The design of *FastSwap* is original in three aspects:

- *FastSwap* provides an efficient and transparent memory sharing mechanism by creating a shared memory based swap area between the host and its VMs. This shared memory mechanism can effectively leverage idle memory present in the host or the other VMs on the host and speed up the VM memory swapping performance by minimizing or avoiding the high overhead of disk I/O for memory swapping on guest VMs.
- *FastSwap* improves both memory swap-out and swap-in performance by introducing a suite of shared memory optimizations. (1) *FastSwap* compresses swap-out pages before placing them in the shared memory or disk swap area and decompresses swap-in pages before making them available for VM execution. (2) *FastSwap* supports four different compression ratios to group compressed pages into four compressed page groups: 512B, 1KB, 2KB and 4KB. (3) *FastSwap* implements a compressed swap page table as an efficient index structure to enable fast lookup of compressed pages in four granularity groups from its swap partitions. This index enables high compression ratio and fast VM swapping for many big data applications.
- *FastSwap* provides a hybrid swap-out scheme to handle the situation of limited shared memory for VM memory paging, which redirect memory swapping to shared memory first and then managing the remaining swapping out traffic on the slower external backing storage such as disk swap partition. In addition, *FastSwap* provides a two-level proactive batch swap-in facility to speed up the performance of VM memory swap-in operations in the presence of sufficient shared memory.

The first prototype of *FastSwap* is implemented on KVM platform [92]. We evaluate the performance of *FastSwap* using a set of big data analytics applications and bench-

marks, such as Redis, Spark, HiBench, SparkBench and YCSB. We show that compared to conventional VM memory management in conventional virtualized Cloud, which allocates memory at VM initialization and shares nothing during VM execution, *FastSwap* can effectively and transparently improve application performance by creating and managing shared memory among hosted VMs. For example, *FastSwap* can speed up the Redis execution throughput by **516x** and improves the Spark runtime by **8.73x** by leveraging unused host memory present in other VMs when the applications are experiencing high memory pressure. *FastSwap* is open sourced on https://github.com/git-disl/FastSwap. The initial test results from several external collaborators show consistent performance gains across both big data driven No-SQL systems and big data driven machine learning workloads.

3.2 Design Overview

The system architecture of *FastSwap* consists of four core functional components as shown in Figure 4.3:

- **Dynamic Shared Memory Management.** A piece of shared memory is pre-allocated in the host and shared by multiple VMs as their swap areas. This shared memory is equally divided into small chunks. Each chunk is dynamically assigned to and revoked from a VM based on its swap traffic, enabling dynamic shared memory consolidation.
- **Hybrid Swap-out.** When the amount of shared memory can not hold all the memory pages swapped out from a VM, *FastSwap* uses shared memory swap area as the fast primary swap partition and resort to the secondary storage swap partition for least recently swap-out pages. This functionality improves the overall VM swapping performance even when a small amount of host idle memory can be leveraged as the shared memory swap area.
- **Two Level Proactive Swap-in.** *FastSwap* improves page fault triggered swap-in performance by two level proactive batch swap-in: (i) the first level is a threshold controlled batch swap-in scheme, enabling VM to proactively swap-in all pages or a threshold of

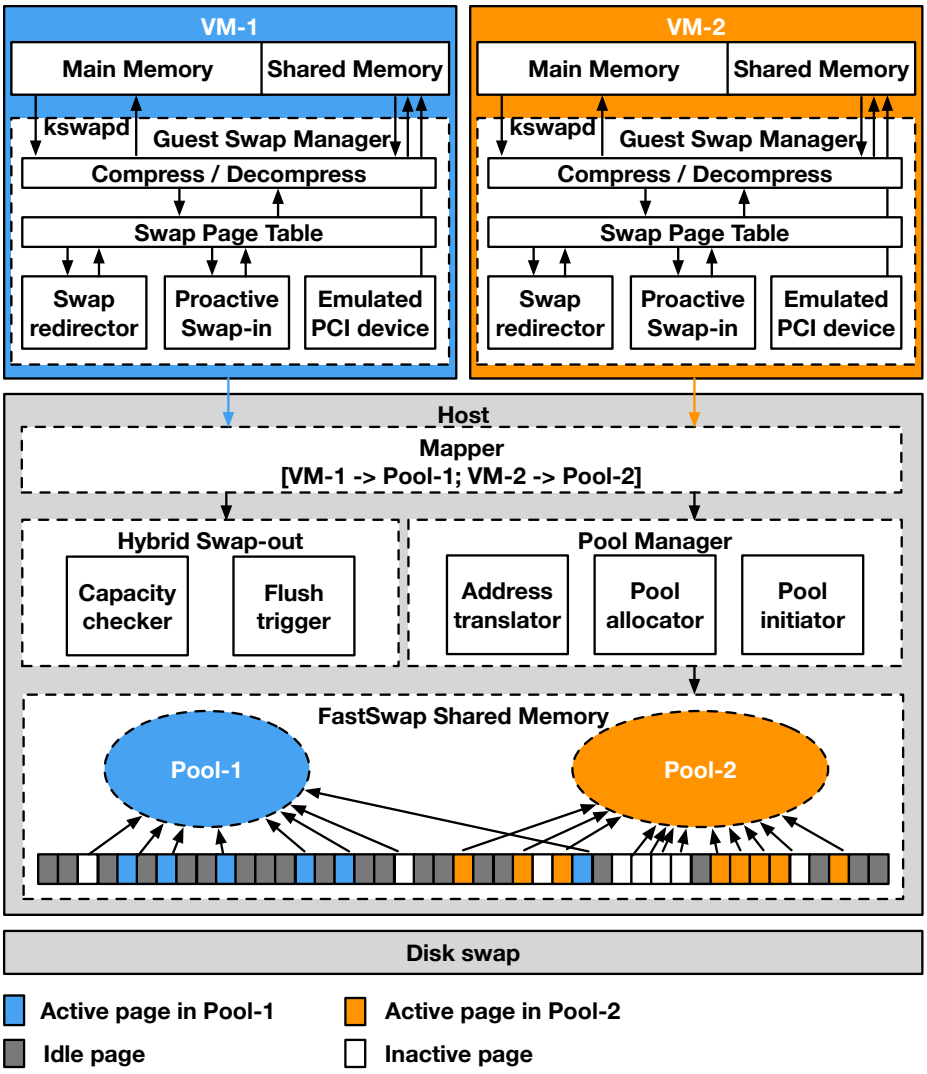


Figure 3.3: *FastSwap* System Architecture

pages as a batch from the shared memory swap partition to the VM main memory; and (ii) the second level is an opportunistic batch swap-in scheme, which minimizes the costly page faults and enables fast recovery of application performance when sufficient memory is made available to the VM under memory pressure.

- **Multi-granularity Compression of Shared Memory.**

  FastSwap by design compresses all swap-out pages before placing them into the proper swap partition. Similarly, all swap-in pages will be decompressed first before returning them to the application's working memory. To further improve the shared memory utilization, *FastSwap* designs an effective multi-granularity scheme for compression of memory

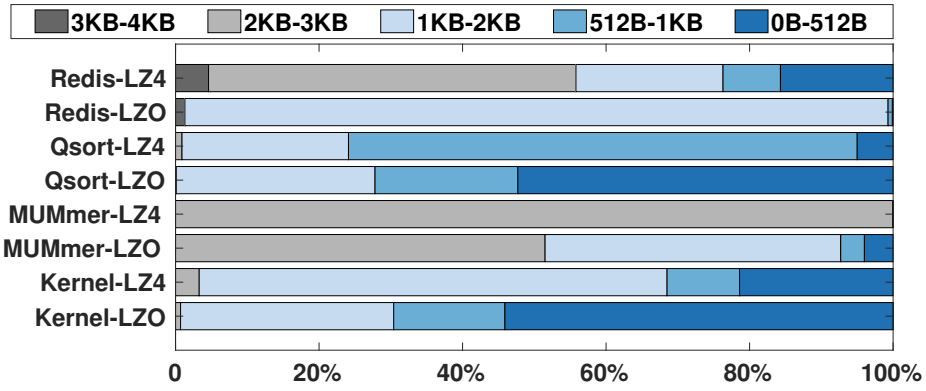


Figure 3.4: Data Granularity

swap pages. This is motivated by the argument that the efficacy of compression approach mainly depends on two factors, compressibility of data and choice of data granularity [93, 94].

Figure 3.4 shows the compression of the memory pages using two popular compression algorithms LZO-1X [95] and LZ4 [96] for four applications: Linux 4.1.0 kernel compilation, MUMmer 3.0, QuickSort algorithm with 20 million random number and Redis 3.0.7 with random workload generated by YCSB. We observe that the percentage of memory pages in different compression size groups varies significantly by the two compression algorithms and also varies from application to application, leading to different compression ratios. These observations motivate us to carefully select a memory compression algorithm with high compression ratio and to design a multi-granularity swap page compression scheme.

There are two alternative ways to implement the host-coordinated shared memory swapping: (i) swapping to host ramdisk and (ii) swapping to host-guest shared memory.

Ramdisk based Swapping builds a ramdisk for a VM in the host memory and mounting this ramdisk as the swapping area of the guest VM. Each ramdisk can be mounted to a different VM at different times based on the swapping demands of the guest VMs. The ramdisk approach is simple and requires no guest kernel modification. However, ramdisk solution has some inherent performance limitations. Although the ramdisk is residing in

the host memory, it is still used by the guest VM as a block device. As a result, certain overheads that are applicable to disk I/O also apply to ramdisk I/O. For example, a disk I/O request from the application running on a guest VM will first go through the guest OS kernel before the request and the I/O data are copied from the guest kernel to the host OS kernel. Hence, when a guest VM swaps its memory pages to a mounted ramdisk, it suffers from two major sources of overhead: the frequent context switch and the data copying between the host user space and host kernel space.

Shared Memory based Swapping is designed and implemented in *FastSwap*: a memory region provided by the host is mapped into the guest VM address space and used as the host-guest shared swapping area for the guest VM. As a part of the system initialization, a shared memory region is divided into multiple elastic memory *pools*, with each corresponding to a specific VM. Each pool is again divided into slabs with a pre-configured slab size. A pool manager at host performs two main tasks: (i) maintaining the mapping between the page offset in the VM swap area and the address in the corresponding shared memory pool, and (ii) dynamically adjusting the size of each VM pool by allocating or deallocating a slab from the host shared memory. The pages in the shared memory are categorized into three types: *active* pages refer to ones being used by the VMs as their swapping destination, *inactive* pages are those allocated to some pool but has not being used yet, and *idle* pages indicate those shared memory pages that do not belong to any pool. The *kswapd* as a default kernel daemon is responsible for memory page swap-in and swap-out. The swap redirector intercepts and redirects the swap in/out pages from/to the VM shared memory area. The *FastSwap* shared memory approach delivers up to 1.7 times faster performance than the Ramdisk solution on Redis, Spark and several benchmark workloads [97] without any modifications to user applications and the OSes.

3.3 Memory Swap Page Compression

In the first prototype of *FastSwap*, we maintain four *fastswap _pool*s to support four compressed page groups, each is represented by a linked list with the *Pool Header* as the start node of pool, and each member of the list corresponds to a shared memory page. For example, the *BUD Header* that belongs to 512B pool will manage a 4K page that has 8 slots to store eight 512B compressed pages. Every time we put one compressed page in, the available slot reduces by one and the occupied slot is recorded. When there is no more *BUD Header* in the free list, a new *BUD Header* will be created and one free shared memory page will be assigned to it. *FastSwap* has a global pointer that keeps track of all used shared memory pages. *Pool Header* contains two pointers, one points to full list and the other points to free list. We maintain an LRU list for each pool. Whenever one *BUD Header* is updated, such as adding or deleting a slot, this *BUD Header* will delete itself from the current LRU and re-insert itself to the beginning of this LRU list. By maintaining four pools separately, four *fastswap_pools* can work concurrently to improve the throughput of VM memory swapping.

A *FastSwap* store operation is invoked when a page is selected for swapping-out by the guest OS. *FastSwap* intercepts the page using the *swap_writepage()*. The operation begins by compressing the page into a temporary buffer in order to know which one of the four compressed page-groups this compressed page should belong to. Based on the compressed size, *FastSwap* chooses one of the four compression granularity pools, which is most suitable for holding the compressed swap page. If the free list in the pool is not empty, then we get the free slot address by reading *BUD Header* meta data. Otherwise, allocating a new *BUD Header*, linking it to the free list, assigning it a new free shared memory page, and have temporary buffer copied into the corrected slot address.

A *FastSwap* load operation is invoked when a page fault is triggered to swap-in a page with a page table entry (PTE) that contains a swap entry. This page-fault based swap-in

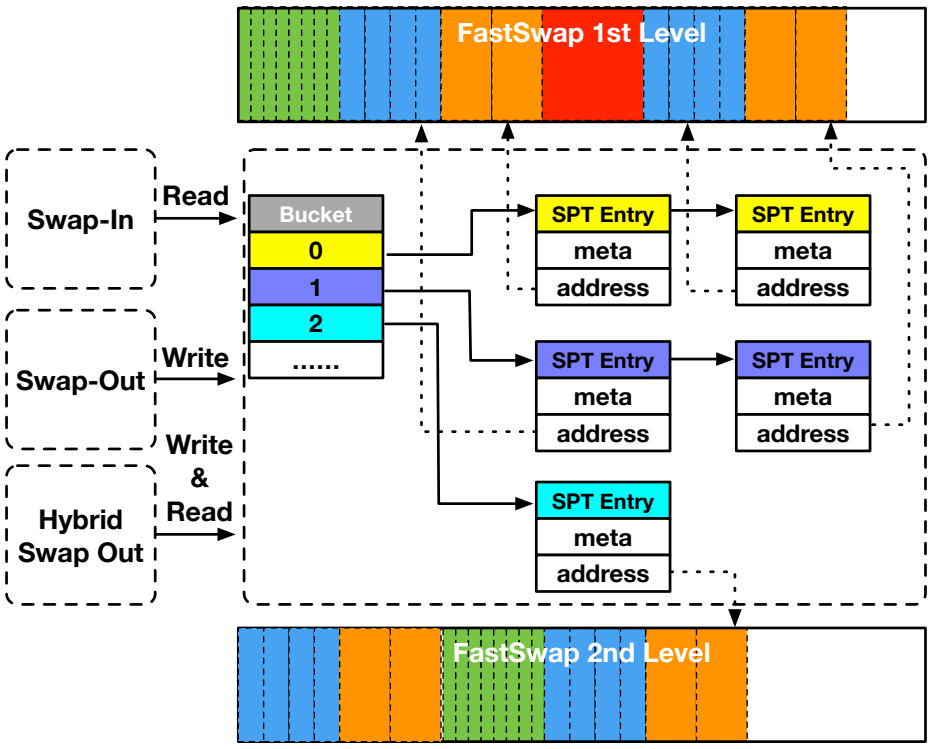


Figure 3.5: Swap Page Compression Index Table

is intercepted by *FastSwap* in *swap_readpage()*. The swap entry contains the page information needed to look up the *swap_page_table_entry* in Swap Page Compression Index Table. Once the entry is located, the address of that compressed page can be read by *swap_page_table_entry*. Finally, the data is decompressed directly into the page allocated by the page fault event and the *BUD Header* containing the page updates its LRU status by putting it to the top of the LRU list (most recent one). When the shared memory is insufficient, *FastSwap* invokes the hybrid swap out algorithm (see Section 3.4 for detail).

To support efficient lookup of compressed swap pages, we create and maintain the Swap Page Compression Index Table (SPCIT) with meta data about each compressed page, and the address mapping between VM memory page address and its compressed page address using Hashtable, which can be built by *hlist*, a kernel linked list data structure in Linux. In the first prototype, we implement the Hashtable as an array of struct *hlist_head* pointers, where each points to a different list, and each list holds all elements that are hashed to the same bucket. Thus, every element is essentially a member of a *hlist* and the Hashtable only

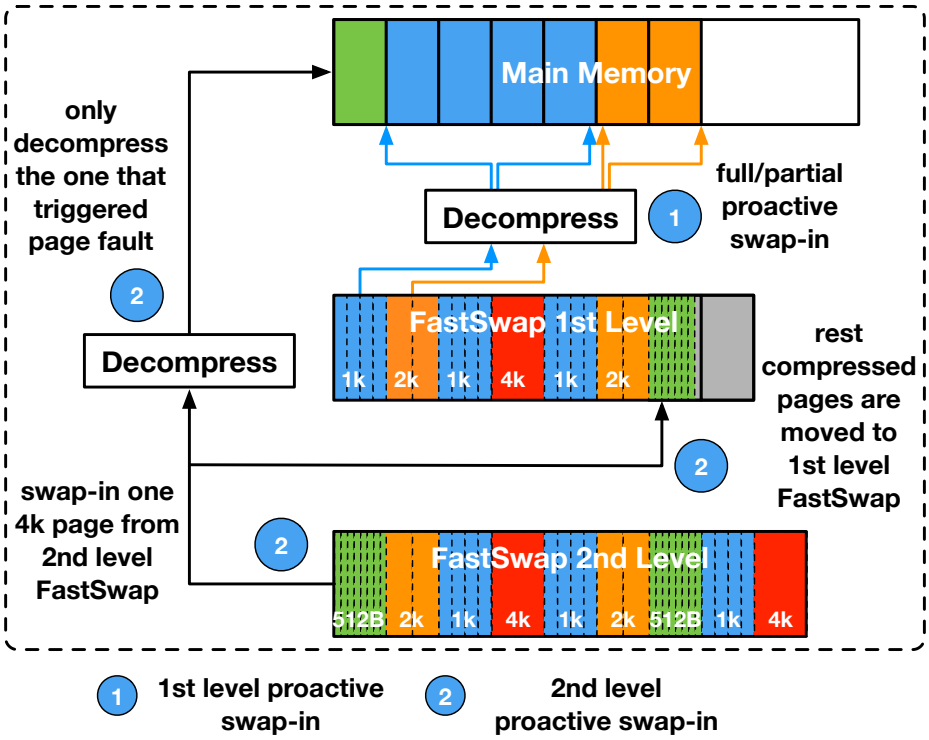


Figure 3.6: Two Level Proactive Swap In

holds the head of these lists. When *FastSwap* intercepts a swap out page request, SPCIT creates a *swap_page_table_entry* for the swapped out page and links it to the Hashtable (see Figure 3.5). Within that entry, a binary variable field is used to indicate the location of the swap partition where the compressed page currently resides: either 1st-level shared memory or 2nd-level external backing storage (e.g., disk or remote memory). We also record in the *swap_page_table_entry* other metadata, such as the address of this compressed page, and the length of compressed page which is used for decompression. We also add *spt_chain* in each entry for handling hash collision. Similarly, the metadata structure used to implement proactive swap-in is also stored in the *swap_page_table_entry*, such as the address of the PTEs corresponding to this compressed swap page.

3.4 Hybrid Swap-out

Hybrid Swap-out is designed to make the shared memory as the fast primary swap partition

with smooth transition to the external backing swap storage as the secondary swap partition when the size of shared memory is not large enough to hold all the swapped pages from a VM. We design an optimal hybrid swap-out model to keep most recent swap traffic in shared memory by moving (dumping) least recent swap-out pages to external swap device (e.g., disk). *FastSwap* implements *least recent compressed swap pages to the secondary swap partition* approach, which uses a user-defined system default batch dump threshold, such as 80%, as the default configuration. When the shared memory swap area reaches 80% full, it will flush all pages in the shared memory swap area to the secondary swap partition by batch I/Os. This approach avoids frequent LRU sorting, decompression latency, and per-page based eviction. Alternatively, when the shared memory is full, the newly swap-out pages will be written to the disk swap area. This approach is straightforward but naive and suffers from poor performance because most of the pages stored in the shared memory are older and has the worst utilization of shared memory and reduces its performance to the conventional OS memory paging as soon as the shared memory swap is filled. By the principle of locality, programs, especially big data analytics jobs, tend to reuse data near those they have used recently.

3.5 Two Level Proactive Batch Swap-in

The idea of proactive swap-in promotes the opportunistic swap-in pages in batch by supporting the first level swap-in from shared memory and the second level swap-in from external swap backing storage such as disk. First, we group a portion of the swapped-out pages in the shared memory swap area into a swap-in batch and proactively perform the batch swap-in without guest memory paging. We implement a threshold-based batch swap-in. By using the threshold of 100% all the pages that are previously swapped out to the shared memory swap partition will be grouped into one swap-in batch. This design allows one to configure and tune the swap-in batch size based on workload characteristics as well as VM available memory.

Table 3.1: LR Performance Speedup w. Diskswap as the Baseline (m:millions)

	Dataset	DiskSwap	ZSwap	FS-baseline	FastSwap
T-1	5m/18GB	1	1.46	2.2	**2.2**
	5.5m/20GB	1	crash	**2.37**	2.29
	6m/22GB	1	crash	**2.43**	2.35
T-2	5m/18GB	1	1.10	1.35	**1.38**
	5.5m/20GB	1	crash	2.55	**2.58**
	6m/22GB	1	crash	**8.73**	8.35

For example, if a page previously swapped out to disk is being accessed by guest VM via a page fault, then a 4KB disk block containing the requested swap-out page is fetched from the disk swap partition. We first locate the requested page in compressed format by lookup in SPCIT and decompress it before moving it to the VM memory. In addition, if the VM has sufficient memory, we will proactively move those remaining compressed pages contained in the same disk block as the requested swap-in page to the shared memory swap partition without decompression. Finally, we update the locality of these pages from disk swap partition to shared memory swap partition. We maintain some metadata in SPCIT, which can assist *FastSwap* to quickly locate the corresponding PTEs in the page table managed by guest OS. For example, when a page is swapped out, *FastSwap* will keep the address of the PTEs related to this page together with the swapped-out page. The address of the corresponding PTE is kept as the metadata in *swap_page_table_entry*. For each page of 4K bytes, this metadata only takes up 4 bytes. The cost of keeping this metadata in the swap area is only about 1/1000 of the total size of the swapped-out pages. For those shared pages which have multiple PTEs, we allocate a specific area in the shared memory as *PTE store*. In this case, the first byte of the metadata specifies the number of PTEs related to this page, while the lasts three bytes is an index pointing the first related PTE in the *PTE store*. This allows *FastSwap* to quickly locate the PTE(s) that needs to be updated by referring without scanning the page table when a page is swapped in. More importantly, the time spent on accessing the PTE of a swapped page will not increase as the size of the system wide page table grows.

Table 3.2: Comparison Results of LR and Redis with Two Experimental Setups: T-1 and T-2

	Metric	DiskSwap	ZSwap	FS-baseline	FastSwap
T-1	Avg. CPU iowait(%)	56.86	60.29(106%)	**16.89(30%)**	19.98(53%)
	Avg. swap-in(page/sec)	54.44	62.55(115%)	**1783.98(3277%)**	1657.86(3045%)
	Avg. swap-out(page/sec)	1856.43	521.58(28%)	**7003.46(377%)**	6912.81(372%)
	Total swap space (MB)	6788.29	11864.18(175%)	6514.11(96%)	**4372.12(64%)**
	Avg. Redis Throughput(ops/sec)	46.24	1100.77(2381%)	**23898.79(51600%)**	20913.04(45227%)
T-2	Avg. CPU iowait(%)	61.29	58.89(96%)	18.78(31%)	**12.37(20%)**
	Avg. swap-in(page/sec)	598.05	221.28(37%)	**15816.58(2644%)**	14510.55(2426%)
	Avg. swap-out(page/sec)	781.88	779.75(100%)	**19460.07(2589%)**	16471.14(2107%)
	Total swap space (MB)	8159.05	7910.57(97%)	8344.68(102%)	**4689.39(58%)**

3.6 Evaluation

We evaluate *FastSwap* using popular big data benchmarks. All experiments are conducted on host server with eight-core Xeon-E5640 CPU, 96GB memory and a SCSI disk, running RedHat Enterprise Linux 6.8 (kernel 2.6.34) and using KVM 1.2.0 with QEMU 2.0.0 as the virtualization platform. The guest VMs run Ubuntu 14.04 with kernel version 4.1.0.

Five typical data analytics benchmark applications in Spark are used in the experiments: *LogisticRegression* (LR), *KMeans*, *Bayes*, *NWeight* (NW), *PageRank* (PR). For LR, we use the datasets produced by SparkBench [98] with varying sizes of samples from 2, 3, 4, 5, 5.5 to 6 millions, which corresponds to different storage sizes of approximately 12GB, 14GB, 16GB, 18GB, 20GB and 22GB respectively. Datasets used in KMeans, Bayes, NW and PR experiments are generated by HiBench. KMeans uses 20-dimension feature vectors with 35 million samples. Bayes uses 2.5 million pages with 100 classes and n-gram 2. NW dataset has 4 million edges with minimum out-edges of 3 and max out-edges of 30. PR dataset includes 750,000 pages. The maximum JVM heap size of each executor is set to be 28GB. In addition to the five types of Spark analytics workload, Redis is used as memory intensive database workloads. The size of the datasets randomly generated by YCSB varies from 10GB to 30GB. We compare three different memory paging alternatives: (1) *FastSwap*, a compressed shared memory swapper with multi-granularity compression, hybrid swap-out, and two-level proactive swap-in; (2) *DiskSwap*, a conventional OS disk swap facility. We refer to this setup as the baseline. (3) *ZSwap* [99], a compressed RAM

cache for disk based swap devices. Swap-out pages are first compressed and maintained in the reserved RAM swap area. When the RAM is exhausted, each swap-out page operation will trigger an eviction from the RAM by finding the least recently used (LRU) page and decompress it and send it to disk swap partition. *ZSwap* can be viewed as a naive baseline implementation of our shared memory optimization.

3.6.1 Overall Performance of *FastSwap*

We use all five types of Spark analytics workloads and the in-memory key-value database benchmark − Redis to evaluate the effectiveness of *FastSwap*.

Performance Comparison when Applications Compete Memory Resource on a VM (T-1 case). We consider the scenario in which two memory intensive applications are running in parallel on one VM and the host has sufficient unused memory since other VMs are idle. We use Spark LogisticRegression (LR) workload and Redis with YCSB read workload. LR is a representative machine learning application that performs iterative computations. It first loads and caches the entire training dataset into the main memory, then iteratively compute the logistic regression and updates the model until the pre-defined convergence condition is met. In all experiments, we run 3 iterations by default unless explicitly stated otherwise. YCSB read-all workload is used to simulate massive data access, with the zipfian distribution. For *DiskSwap* solution, we setup a Redis in-memory database server on a VM with 30GB main memory and pre-load 10GB key-value data to the VM, and then start LR and YCSB read-all workload together. For LR workloads, we use datasets of 18GB, 20GB and 22GB, representing the three scenarios in which the LR dataset can fit in memory with 100% or 75% or 50%. We measure the LR execution time, the Redis throughput, and system resource utilization. We report the measurement results in the T-1 section of Table 3.1 and provide the detailed analysis measurements in the T-1 section of Table 3.2. Table 3.1 T-1 section displays the speedups achieved by *FastSwap* without compression (*FastSwap baseline*) are ranging from 2.2x to 2.43x, and by *FastSwap*

are ranging from 2.2x to 2.35x. In comparison, ZSwap crashes in the second and third LR test (20GB, 5.5 million samples and 22GB, 6 million samples), because the long latency of VM swapping causes the Spark executor heartbeat time out.

The detailed system level measurements are given in the T-1 section of Table 3.2. We make several interesting observations. *First*, compared to DiskSwap case, *FastSwap* and *FastSwap* without compression (*FastSwap baseline*) significantly reduces the CPU iowait by 47% (1-53%) and 70% (1-30%) respectively. It shows that *FastSwap* increases swap-in speed by more than 30x and swap-out speed by 3x, thus effectively minimizing the latency of VM swapping. In comparison, ZSwap incurs higher CPU iowait (106%) than the traditional disk swap solution. ZSwap performs poorly is due to its per-page based swap-in and swap-out solution combined with the cost of making LRU eviction decision for every page to be swapped in or swapped out after the shared memory is full, in addition to the cost of frequent compression/decompression. *Second*, *FastSwap* reduces the shared memory consumption to 64% due to the use of multi-granularity of compression groups. Furthermore, by comparing *FastSwap* and *FastSwap* without compression, we observe that the compression incurs very small overhead. *Third*, *FastSwap* and *FastSwap* without compression show a significant throughput performance boost for Redis workloads with 452x and 516x respectively, compared to the performance of the traditional DiskSwap solution. More interestingly, *FastSwap* can improve both Spark workloads and Redis workloads at the same time due to shared memory swap optimization, even when both applications are competing for the scarce physical memory resource.

Performance Comparison under Multiple VMs with Memory Contention (T-2 case). This set of experiments focuses on measuring the effectiveness of *FastSwap* when two VMs both run memory intensive big data applications. We setup two VMs with each allocated 30GB main memory. VM1 runs Spark with LR workload and VM2 runs Redis with YCSB load workload. The Redis client loads 30GB key-value data generated by YCSB to the Redis server running on VM2, which causes high memory pressure on VM2. Given that

VM1 is initialized and have unused memory, by using the balloon driver, we move 10GB from VM1 to VM2 [31]. With 40GB running Redis workloads of 30GB and 20GB memory remaining on VM1 to run Spark LR workload of three sizes. The performance results are shown in the T-2 section of Table 3.1 and Table 3.2. Compared to the traditional *DiskSwap*, both *FastSwap* and its baseline (without compression) respond well and provide much better performance in the presence of the increasing memory contention on VM1 due to the increased size of the training datasets from 5 million (18GB) to 5.5 million (20GB) and 6 million (22GB). However, ZSwap crashed again for the datasets of 20GB and 22GB. For 22GB dataset, *FastSwap* and *FastSwap* without compression provides 8.35x and 8.73x speedup respectively over DiskSwap. Also compared to *FastSwap* baseline, *FastSwap* offers significant reduction of swap storage (58%) and the smallest CPU on iowait (20%) compared to both *DiskSwap* and *FastSwap* baseline. This shows that multi-granularity compression in *FastSwap* incurs low overhead, with slightly slower performance than *FastSwap* without compression for most benchmarks.

Performance under Multiple Benchmarks. In the next set of experiments, all five types of Spark analytics workloads are used with the same experimental setup. Table 3.3 shows the results. We observe that *FastSwap* (and its non-compression baseline *FS-baseline*) significantly outperforms DiskSwap and ZSwap for all 7 benchmarks. For example, *FastSwap* can run the LR with 6 million samples (22GB) at only 11.9% (11.5 mins) of the time spent by using the DiskSwap solution (96 mins). Also when using DiskSwap, two Spark benchmarks (Bayes and PageRank) crashed, and when using ZSwap, five Spark benchmarks (KMeans, Bayes, PageRank, LR 20GB and LR 22GB) crashed. This is due to the Spark heartbeat time out because of the poor performance of VM swapping when using the conventional DiskSwap and ZSwap.

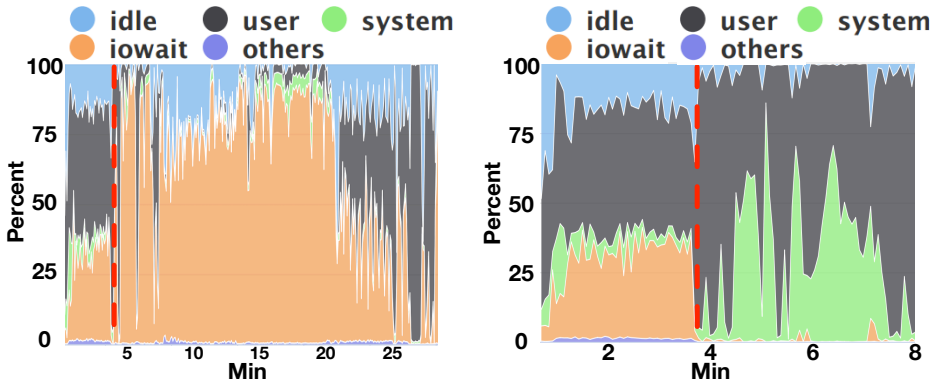
To further analyze the effectiveness of *FastSwap*, Figure 3.7 and Figure 3.8 show the CPU usage and disk throughput for KMeans workloads respectively. Similar with LogisticRegression, KMeans first loads and caches into the VM main memory the training

Table 3.3: Execution Time (mins) of Benchmarks (m:million)

	DiskSwap	**ZSwap**	**FS-baseline**	**FastSwap**
KMeans	28	crash	8	8.3
Bayes	crash	crash	13	13
NW	25	4	2.8	3
PR	crash	crash	4.5	4.7
LR 5m	11	10	8.1	8
LR 5.5m	24	crash	9.4	9.3
LR 6m	96	crash	11	11.5

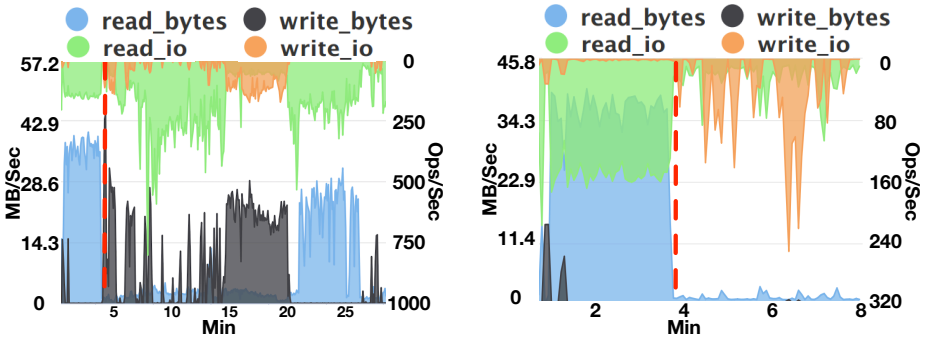
dataset of 35 million samples with dimension-20, generated by GenKMeansDataset based on Guassian distribution, then it iteratively updates the model until the pre-defined convergence condition is met. Thus, KMeans workload consumes huge amount of memory. The red dotted line in Figure 3.7 and Figure 3.8 indicates the starting point of VM swapping due to insufficient physical memory. Given that DiskSwap finishes KMeans workload in 28 mins, the x-axis varies up to 28 minutes. Similar setup of x-axis for *FastSwap*. For ZSwap, it did not finish after executed for 35 minutes and crashed eventually for the KMeans workload. Thus, we omit ZSwap in this comparison.

We make several observations. *First*, Figure 3.7a shows that DiskSwap incurs high CPU utilization on iowait during swapping. Figure 3.8a shows large disk write operations due to memory swapping to disk. In comparison, *FastSwap* spent most of the CPU on shared memory swapping and little CPU on iowait and little disk writes in the presence of VM swapping. We also observe that the average of CPU for system usage for *FastSwap* without compression is slightly lower than that of *FastSwap*. This is mainly due to the CPU cost for compression in *FastSwap*. Table 3.3 shows that the compression overhead has negligible overhead on the execution time for all 7 benchmarks. *Third*, DiskSwap has large percentage of CPU spent on iowait, due to slow disk write operations during swap in and out.



(a) DiskSwap (b) FastSwap

Figure 3.7: CPU Usage of KMeans

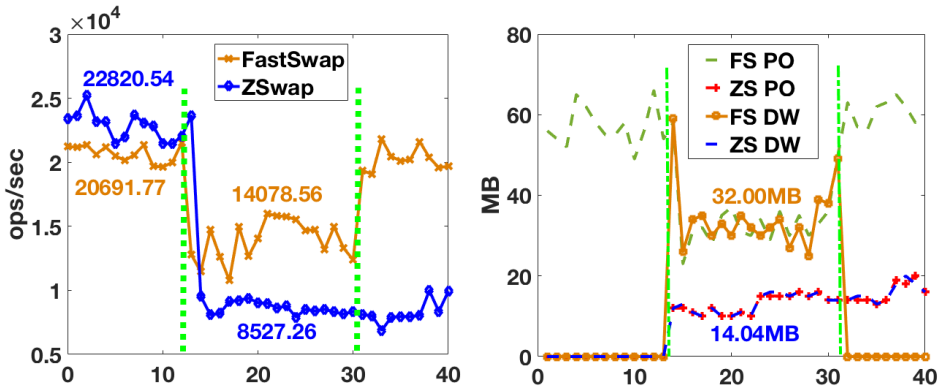


(a) DiskSwap (b) FastSwap

Figure 3.8: Disk Throughput of KMeans

3.6.2 Hybrid Swap-Out

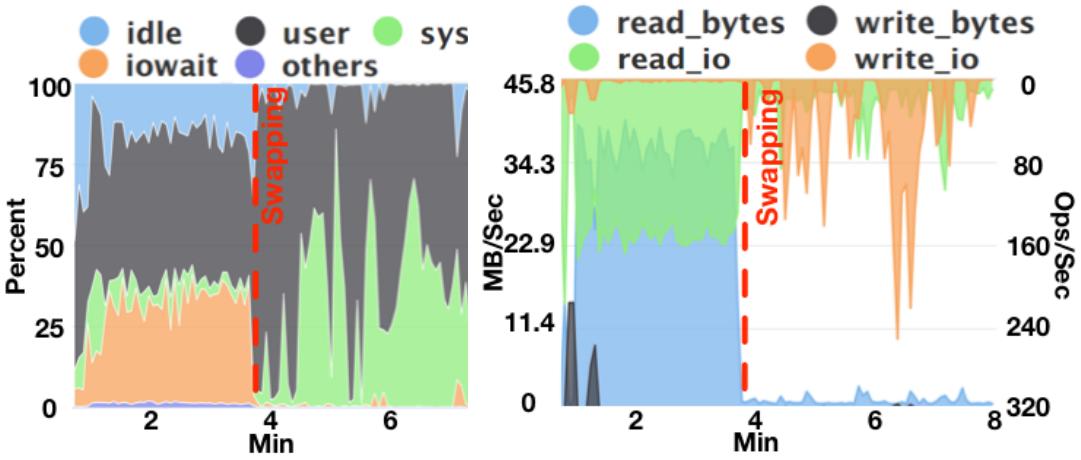
In this section, we evaluate the effectiveness of the hybrid swap-out mechanism in *Fastswap* by considering the cases when the shared memory swap space is insufficient to hold all the pages swapped out from the guest VM. We measure the performance of Redis and Spark workloads, when the shared memory swap area is 80% full (or reaches a pre-defined dump threshold). In this case, *FastSwap* performs the hybrid swap-out by copying pages from the shared memory swap partition to the disk swap partition. For the sake of comparison, we use a similar setup for this set of experiments and measure both *FastSwap* and ZSwap. Figure 3.9 shows the Redis throughput (ops/sec) and the disk I/O in MB. The two vertical green dotted lines indicate the period of hybrid swap-out dumping process. The first dotted



(a) Redis Throughput (b) Disk Throughput

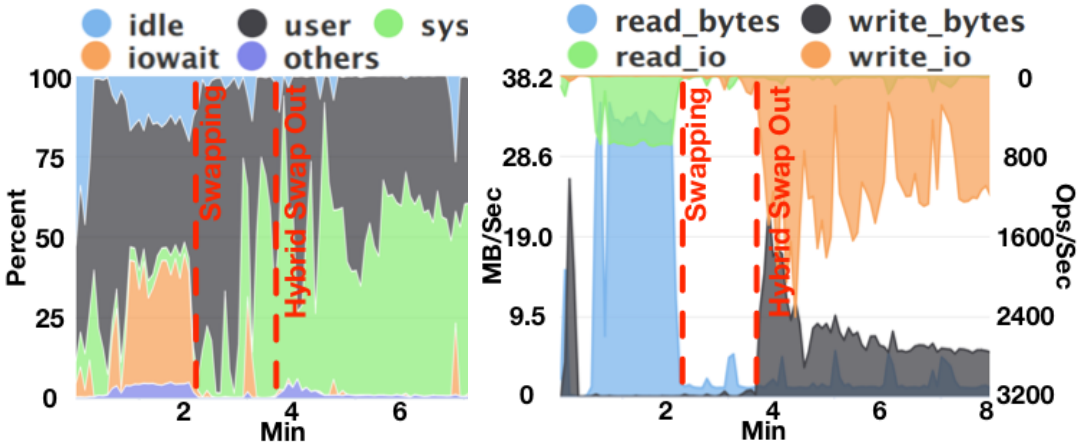
Figure 3.9: Redis Performance with Hybrid Swap-Out.

line is also the time when ZSwap physical memory capacity is full, because we choose the dump threshold in *FastSwap* accordingly for fair comparison. We make three observations from Figure 3.9. *First*, during the hybrid swap-out process, Redis throughput is degraded by 32% with *FastSwap* but with ZSwap, Redis throughput is degraded by 63%. Also the throughput of Redis with *FastSwap* consistently outperforms that with ZSwap during the entire hybrid swap-out dumping period. *Second*, as soon as *FastSwap* dumping process completes, Redis restores to its best throughput performance. In contrast, using ZSwap, Redis is running with much lower throughput and remains at the poor throughput even after the hybrid swap-out dumping period, due to the lack of optimizations in ZSwap design. For example, in ZSwap, when its swap RAM cache is exhausted, each of its swap-out page operations will trigger an eviction from the RAM by finding the least recently used (LRU) page and decompress it and send it to disk swap partition. This design incurs high overhead due to per-page based swap in and out, combined with the cost of compression/decompression and the cost of making LRU eviction decision for every swapped page after its swap RAM area is full. *Third*, in contrast, *FastSwap* keeps pages that need to be dumped to the disk swap area compressed by leveraging our compressed swap page table. Such design efficiently saves I/O bandwidth and shortens the hybrid swap-out dumping period. Figure 3.9b displays the average disk write speed during the hybrid swap-out dumping



(a) CPU Usage (b) Disk Throughput

Figure 3.10: KMeans Performance w.o. Hybrid Swap-Out.



(a) CPU Usage (b) Disk Throughput

Figure 3.11: KMeans Performance with Hybrid Swap-Out.

process, which is 32MB/sec (compared to 14.04MB for ZSwap) and the dumping process lasts for 18sec. Also the size of data being dumped is 576MB, which is only 56% of 1GB, thanks to the *FastSwap* compression design. Finally, we would like to comment that in the first 10 seconds, the Redis has higher throughput with ZSwap than with *FastSwap*. The reason is primarily related to the performance difference between physical memory and shared memory. ZSwap utilizes physical memory and *FastSwap* utilizes shared memory to store pages.

Next, we evaluate the performance of Spark KMeans workloads during the hybrid

swap-out process. We measure CPU usage and disk throughput to compare FastSwap-without-hybrid-swap-out with FastSwap-with-hybrid-swap-out as shown in Figure 3.10 and Figure 3.11 respectively. The hybrid swap-out is triggered when the default threshold of shared memory swap partition is reached (e.g., 80%). The first red dotted line in Figure 3.10 and Figure 3.11 indicates the start point of guest VM swapping and the second one indicates the starting point of hybrid swap-out. A number of observations can be made. *First*, the execution time of FastSwap-with-hybrid-swap-out is 8.2 mins and that of FastSwap-without-hybrid-swap-out is 8 mins, which means hybrid swap-out only causes 2.5% performance degradation. Such overhead mainly comes from two sources: (i) reading pages that had been moved to the disk swap partition; and (ii) additional CPU cycles used by the hybrid swap out thread. *Second*, during the hybrid swap-out process, *FastSwap* did incur large amount of disk write operations as shown in the dark area of Figure 3.11b. However, these write operations do not increase CPU iowait, indicating good disk I/O efficiency for writes. *Third*, since a separated thread is created in kernel space only for the hybrid swap out purpose, from 2.2 mins to the end in Figure 3.11a we can see that this kernel thread is working and consuming CPU percentage on the kernel space operations (i.e., the CPU percentage spent on kernel system).

3.6.3 Proactive Swap-In

Proactive swap-in speed is measured for DiskSwap, FastSwap without compression (FS-baseline), FastSwap level-1 proactive swap-in and FastSwap level-2 proactive swap-in, and the results are shown in Table 3.4. For *DiskSwap*, *FS-baseline* and *FastSwap* level 1 (full swap-in) case, all pages in the shared memory swap area will be swapped in. For *FastSwap* partial case (level 2 proactive swap-in, we let *FastSwap* perform partial proactive swap-in of 500MB repeatedly until there is only 1GB free space left in the main memory.

We make four observations from Table 3.4. (1) Since the process of swapping-in page needs to update PTE, and in *DiskSwap* case, OS needs to scan the page table to find the

Table 3.4: Proactive Swap-In Speed (per page)

	Avg. Time(us)	**s.d.**
DiskSwap	419.66	7.84
FastSwap w.o. compression	5.70 (74x)	0.18
FastSwap (full)	9.56 (44x)	0.30
FastSwap (partial)	9.68 (43x)	0.12

corresponding page table entries that it needs to update, which is very time consuming, the performance of *DiskSwap* case is very slow. (2) Because in *FastSwap* and *FastSwap* without compression case, the OS swaps out not only memory pages but also some meta-data, which can help the OS to quickly identify which page table entries need to be updated during the swap-in process. Thus the cost of scanning the page table is avoided. Both versions of *FastSwap* are a lot faster than *DiskSwap*. (3) Because of the use of compression, *FastSwap* needs to decompress each page before swapping it back into the main memory, leading to higher latency than the version of *FastSwap* without compression. However, comparing with *DiskSwap* case, *FastSwap* is still 44x faster in terms of per-page swap-in performance. (4) The partial batch swap-in option enables small amount of batch swap-in when the hosted VM has insufficient free memory to hold all pages stored in the shared memory swap area. We argue that the full batch swap-in should be treated as a special case of the threshold based batch swap-in to provide high utilization of available VM memory and high utilization of shared memory. From Table 3.4, the partial proactive swap-in displays very similar performance as the full proactive swap-in in terms of per-page swap-in operation time.

3.7 Related Work

Efficient memory management in virtualization platforms is widely acknowledged as a challenging problem for memory intensive workloads. Several orthogonal efforts have targeted at scheduling, allocating and consolidating physical memory among VMs on demand with unmodified guest OS. Ballooning [31] was proposed in 2002. A balloon driver, run-

ning in the guest OS, can help the VM Manager to reclaim or repay guest memory by inflating or deflating the balloon manually. But it relies on administrator to manually when to trigger ballooning and how much to balloon from one VM to another. Although research efforts have been dedicated to periodic estimation of VM working set size, it is well known that accurate VM working set prediction remains to be an open issue under change conditions [8].

VSwapper [91] is a disk-based VM swapping facility and designed to address the uncooperative host swapping problems, such as double paging by tracking the correspondences between disk blocks and guest memory pages and between host and guest VMs. It does not improve VM memory paging performance using shared memory or RAM.

Existing efforts in computer networks community have explored disaggregated memory by memory paging to remote nodes via RDMA network, and Infiniswap [21] is the most recent representative work in RDMA-based remote paging. However, these efforts only support memory swapping to remote host and fail to provide memory sharing across containers/VMs/executors on the same host. Even when there is sufficiently unused memory on the host, VMs or containers have to swap pages to remote host instead.

FastSwap by design can integrate memory sharing across VMs/containers regardless whether they are on the same host and different host machines. This chapter has shown that *FastSwap* offers up to 516x performance improvement for both NoSQL and ML workloads over conventional Linux memory swap facility and 22x improvement over existing open source effort, such as ZSwap, on memory sharing across VMs on the same host. Our preliminary measurement of *FastSwap* shows 14x performance improvement over Infiniswap when integrating RDMA based memory paging to remote nodes when there is insufficient memory on the local host. Due to space constraint, we omit the remote memory sharing techniques and detailed comparison, and refer readers to our technical report for detail [100].

3.8 Conclusions

Big data continues to penetrate our work life and everyday life. Performance of big data analytic workloads continues to dominate the wide deployment of systems, applications and services. We have presented the design of *FastSwap*, a highly efficient shared memory paging facility, with three original contributions. (1) *FastSwap* dynamic shared memory management scheme can effectively utilize the shared memory across VMs through host coordination. (2) *FastSwap* provides efficient support for multi-granularity compression of swap pages in both shared memory and disk swap devices. (3) *FastSwap* provides a hybrid memory swap-out scheme to flush the least recently swap-out pages to disk swap partition when shared memory swap partition reaches a pre-specified threshold and close to full. (4) *FastSwap* provides two level proactive swap-in optimizations (shared memory to memory and disk to shared memory). Our extensive experiments using big data analytics applications and benchmarks demonstrate that *FastSwap* offers up to two orders of magnitude performance improvements over existing memory swapping methods. *FastSwap* will be officially released on github this fall at https://github.com/git-disl/FastSwap.

CHAPTER 4

DYNAMIC HOST AND REMOTE MEMORY SHARING WITH XMEMPOD

This chapter presents *XMemPod*, a disaggregated memory orchestration system that virtualizes cluster wide memory to scale data intensive, large memory workloads in virtualized clouds. *XMemPod* takes a software-defined, application transparent approach with three novel contributions: (1) *XMemPod* dynamically expands the memory capacity of the virtual machines (VMs) under high memory pressure by providing efficient, transparent sharing of unused memory that is disaggregated across VMs on the same host or in the cluster. (2) *XMemPod* provides a hierarchical memory expansion and sharing framework, which enables memory intensive workloads on a VM to expand its memory demand over virtualized host memory first, and remote memory next, before resorting to external disk. (3) *XMemPod* provides a suite of optimization techniques to further improve the utilization and access latency of disaggregated memory. *XMemPod* is deployed on a virtualized RDMA cluster without any modifications to user applications and the OSes. Evaluated with multiple workloads on unmodified Spark, Apache Hadoop, Memcached, Redis and VoltDB, using *XMemPod*, throughputs of these applications improve by 11x to 612x over conventional OS disk swap facility, and by 1.7x to 14x over the existing representative remote memory paging systems, and yet the total amount of network traffic consumed by *XMemPod* is only 24% of the existing approaches.

4.1 Introduction

This chapter describes *XMemPod*, a distributed memory virtualization system, which accelerates bigdata and machine learning (ML) workloads by virtualizing cluster-wide external memory as a low-latency external storage device for memory intensive workloads with transient memory demands. *XMemPod* is designed to dynamically orchestrate terabytes of

cluster wide memory into multiple memory pods. By leveraging guest indirection, *XMemPod* transparently virtualizes cluster-wide unused memory into the physical address space of each guest OS instance on demand. This enables legacy servers to utilize cluster-wide memory for efficient execution of memory intensive workloads that need large datasets to perform low latency I/O (fig. 5.7).

XMemPod is inspired by existing proposals from two orthogonal efforts (§4.7): exploiting non-intrusive sharing of memory between host and its guest VMs [90, 32, 7, 8, 31, 10, 101] and exploiting the disk-network latency gap via unused remote memory [22, 102, 103, 104, 60, 21, 62, 105, 106, 107, 108, 109]. However, unlike existing proposals for dynamic memory balancing, *XMemPod* does not pay any CPU overhead and time delay for memory scanning to detect memory imbalance and does not rely on estimation of the working set size of each VM/container/executor to respond to memory pressure, since accurate estimation of working set size is difficult under changing workloads [91, 90]. Moreover, existing solutions for remote memory paging [22, 15, 21, 106, 107], lack of a holistic approach to provide high performance virtualization of cluster wide memory. For example, they treat the cluster-wide disaggregated memory as one flat tier of external memory resource and fail to take advantage of the I/O latency gap between DRAM and remote network memory.

XMemPod addresses the above challenges by providing two types of disaggregated memory pods, which are made available to unmodified VMs for use as a low-latency in-memory block device, which stores large datasets or serves memory paging events for VM workloads that require more DRAM than its allocation, without any modifications. It first utilizes host-coordinated shared memory pods to take the advantage of performance gap between DRAM and network I/O, in the presence of excess page faults and thrashing, maximizing the utilization of unused host memory. Upon insufficient unused memory on local host, *XMemPod* resorts to remote memory pods, which are configured by reliable partitioning of large cluster wide memory into non-intrusive, elastic remote memory sharing groups with decentralized coordination. *XMemPod* is implemented on Linux kernel 4.1.0

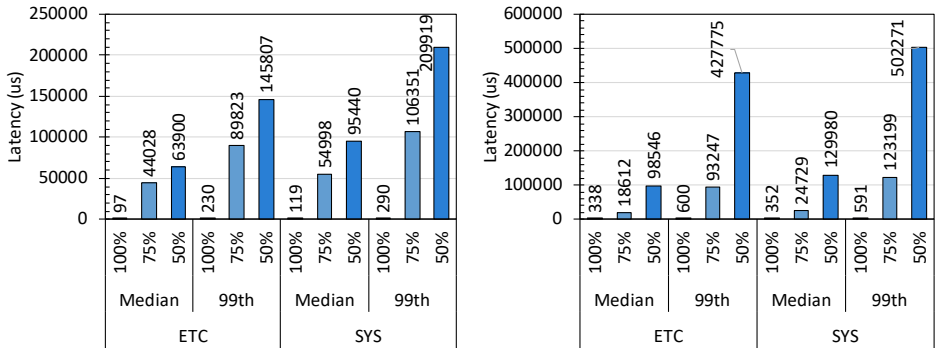
and KVM [92], deployed on a 56Gbps, 32-machine RDMA cluster in Emulab [110] with 2 terabyte (TB) collective memory. We evaluated it using multiple unmodified memory intensive applications: Spark [111, 4], Apache Hadoop [5], Memcached [1], Redis [2], VoltDB [3], and open benchmarks: HiBench [112], SparkBench [98] and YCSB [77]. Using *XMemPod*, throughputs of these applications improve by 11x to 612x, median and tail latencies improve by 174x to 702x and 218x to 630x respectively, over conventional OS swap facility. Compared with the existing approaches such as nbdX or Infiniswap, using *XMemPod*, throughputs improve between 1.7x and 14x, median latencies and tail latencies improve between 1.8x and 12x, and 3.3x and 15x respectively (§4.6).

There are situations that *XMemPod* does not yet handle effectively, including imbalanced usages of host coordinated unused memory shared among its VMs, due to its opportunistic demand driving orchestration policy. We were unable to obtain larger RDMA clusters to perform stress test on scalability.

4.2 Motivation

Memory utilization imbalance and temporal usage variations are frequently observed in virtualized clouds [6, 23, 24, 25, 7, 8, 26, 27, 28], and production datacenters [29, 5, 30, 21, 11, 12]. One study on a google 12K-machine cluster running a mixture of long and short-lived workloads reported around 50% memory utilization, stating "the gap between resource requests and average usage accounted for most of this difference" [9, 11]. Another study on two production datacenters (3,000-machine Facebook analytic cluster and 12,500-machine google cluster) reports severe imbalance in memory utilization for more than 70% of the time across machines [21]. These reports show the potential opportunities of exploiting unused host/remote memory to speed up the performance of VM workloads whose working sets cannot fully fit into their allocated DRAM, and resorting only to disk I/O for persistence, failure recovery and contingence.

Benefits and Adverse Effect of Paging. Modern OS supports the virtual memory ab-



(a) Redis (b) VoltDB

Figure 4.1: The Adverse Effect of Paging

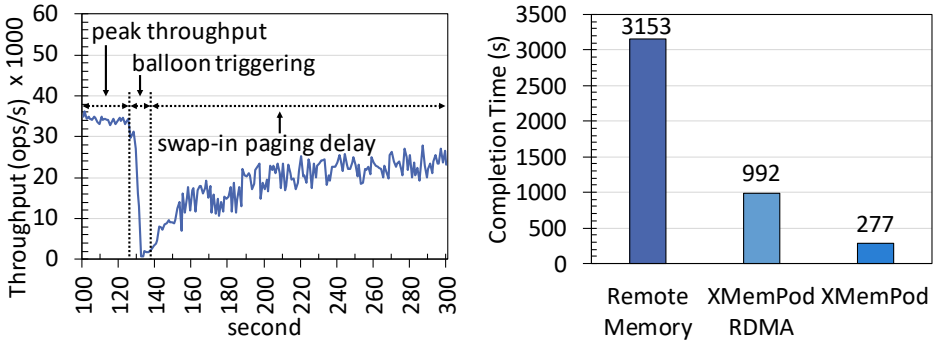
straction (VMA) to provide the extra memory capacity upon memory contention using a secondary backing storage layer, known as swap space. Swap space is typically located on disk, with orders of magnitude slower access latencies (milliseconds) than main memory (nanoseconds). Paging is implemented by VMA manager via swap-out and swap-in operations, triggered by a page fault. VMA works with a block device to access the swap space and serves paging request [113].

We illustrate some adverse effects of paging through two bigdata applications: Redis [2], an in-memory NoSQL store, and VoltDB [3], a translytical in-memory database. We run each application with two benchmark workloads: ETC and SYS (§4.6), using the raw dataset of 20GB from YCSB [77], and measure per server performance. The peak memory that can fit the full working set is measured for both Redis and VoltDB, which are 29GB and 30GB respectively. We run each application on VM with three different memory capacity configurations – 100%, 75%, and 50%. x% indicates that the VM is configured to run an application and can hold x% of its working set in memory. When x $=$ 100%, no paging occurs, but when x $<$ 100%, paging happens, as shown in Figure 4.1. We highlight two observations. *First*, paging has super-linear performance impact on applications. Compared to 100% configuration, when 25% of the working set does not fit in memory (75% configuration), the median latency is worsened by 462x and 70x for Redia and VoltDB respectively. For 50% configuration, another 25% reduction, median latencies are worsened

by 802x and 369x for Redis and VoltDB respectively. *Second*, the adverse effect of paging on 99th percentile latency is consistently larger than its effect on median latencies. For 75% configuration, 99th percentile latencies of Redis and VoltDB are worsened by 391x and 208x respectively, compared to 100% configuration. For 50% configuration, 99th percentile latencies are degraded by 724x and 850x for Redis and VoltDB respectively.

Dynamic Memory Balancing with Balloon Driver. To cope with such harsh performance degradation when the working sets of applications do not fit in memory, memory ballooning [31] was proposed to move unused memory between VMs. However, solely relying on ballooning, applications under memory pressure still suffer hefty performance degradation due to three types of delays: the timing delay of scheduling ballooning, the balloon driver delay for moving sufficient memory, and the time delay for applications to return to their peak performance. Figure 4.2a shows the three types of delays, the performance deterioration incurred, and the adverse effect on application performance, when the working set of Redis using ETC workload (§4.6) does not fit in memory (50% configuration). It is observed that the throughput of Redis deteriorates sharply during the 15 seconds from 123rd to 138th second, due to excessive page-out events, the timing delay of scheduling ballooning and the balloon driver delay for moving sufficient memory. Upon installing sufficient ballooning memory at the 140th second, Redis throughput starts to recover but at a very slow pace. This is because even with additional memory, Redis cannot immediately regain its peak performance due to the slow per page swap-in operation upon each page fault. It takes approximately 140 seconds to fully utilize the newly ballooned memory. Furthermore, ballooning incurs the overheads of scheduling and moving (deflating and inflating) memory from one VM to another on the same host, and cannot leverage unused remote memory in the cluster.

Differentiating External Memory by Latency. In virtualized clouds, multiple VMs run on the same host machine, each having its own memory allocation (typically equal amount). When a VM reaches its memory limit, it has two alternative low-latency memory expan-



(a) Ballooning Delays (b) Comparison of Solutions

Figure 4.2: Alternative Solution Approaches.

sion opportunities before paging to local disk swap partition. They are unused memory on local host and unused remote memory in the cluster. Note that even though SSD is much faster than HDD, it is still two orders of magnitude slower than fast interconnect like InfiniBand based RDMA [64]. At the same time, RDMA interconnect is several times slower than local memory access [65], making latency-sensitive prioritization critical for efficient disaggregated memory orchestration.

XMemPod implements a virtual block device manager, which provides disaggregated memory orchestration service to each VM by leveraging indirection, and interfacing with the host shared memory first, the remote memory next, and resorting only to the disk I/O for exception handling. Figure 4.2b shows the comparison of three alternative design choices using Redis with ETC under 50% cofiguration: (a) using unused memory on the local host solely for serving remote memory paging requests, (b) *XMemPod* RDMA and (c) *XMemPod* which leverages host unused memory first and then remote memory. Using *XMemPod* in case (c), Redis completion time is 11.3x times faster than existing solutions for remote memory paging in case (a), and 3.2x faster than *XMemPod* RDMA in case (b), which treats host unused memory as remote network memory.

4.3 *XMemPod* Overview

XMemPod consists of four core components: the disaggregated memory paging service manager (PSM), the shared memory manager (SSM), the remote memory manager (RMM) and the disk swap partition manager (DSM), as shown in Figure 4.3. They coordinate closely to provide efficient virtualization of cluster wide memory and hierarchical orchestration of disaggregated memory sharing capabilities, including dynamic shared memory allocation/deallocation, multi-granularity disaggregated memory page compression, hybrid page-out and proactive batch page-in optimizations. Client module and server module are the two main functional components of *XMemPod*. A VM that is low on memory will run the *XMemPod* client module to acquire external disaggregated memory service. Every node in the cluster can run the *XMemPod* server module to provide external memory expansion and sharing in response to the requests from *XMemPod* clients in a cluster.

XMemPod is currently implemented on a virtual machine based applications deployment platform, which is common in production datacenters with virtualized clouds [6, 9, 11, 12]. The algorithms and optimization techniques are by design applicable to container based and Spark executor/JVM based application deployment platforms, since all three platforms are deploying application processes by resource and performance isolation via VMs/containers/JVMs. No matter which deployment platform, applications start paging when their working sets no longer fit in memory of their executors. *XMemPod* implements remote memory paging using an Infiniband RDMA network, though it makes no assumption on specific RDMA network technology.

The Memory Paging Service Manager (PSM) is the client module implemented on top of the *kswapd*, a default kernel daemon responsible for memory paging. The PSM intercepts swap-out and swap-in operations from the guest VMA manager and redirects them to *XMemPod*. The PSM provides a virtualized block device interface to the large dataset applications on a guest VM. This virtualized block device can be configured as a low la-

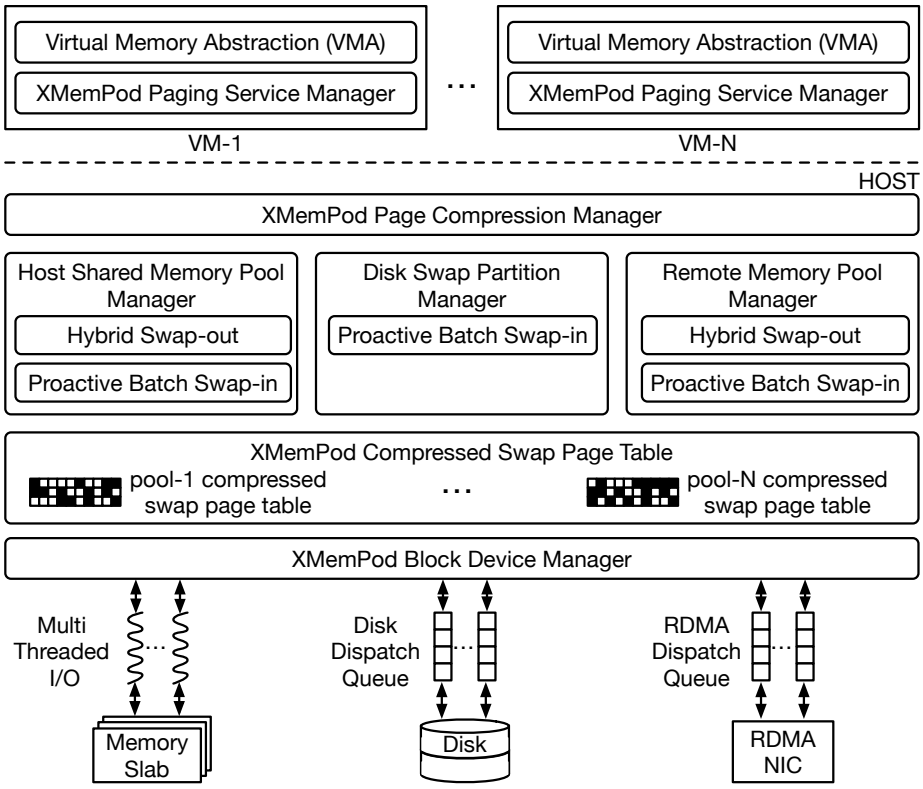


Figure 4.3: *XMemPod* System Architecture.

tency volatile file system for storing large datasets, or a low latency primary swap device, or a memory-mapped I/O space for large memory application. For the guest VM, the block device is simply a fast I/O partition (e.g., a linear I/O space), similar to a regular disk partition, with an order of magnitude smaller access latency than disk. However, internally, the client module maps the single linear I/O space of the block device to the unused memory in either a host-guest shared memory partition or multiple remote shared memory pods distributed across the cluster. *XMemPod* implements the host-guest shared memory swap partition with both ramdisk based option and shared memory based option. The former is mounting a host resident ramdisk to the guest VM, and the latter uses a shared memory pool provided by the host, which is mapped into the host swap partition address space on guest VM. Comparing to the ramdisk solution, the shared memory approach delivers 1.7x better performance via a loadable kernel module for Linux 4.1.0 for all workloads we have tested. By default, *XMemPod* uses shared memory solution for both host and remote

memory sharing. Upon receiving swap-out pages from a client module, the server module (SMM, RMM or DSM) stores the pages either locally in the host-guest shared memory pod or in selected remote memory pods across the cluster.

The Shared Memory Manager (SMM) uses a POSIX shared memory region as the basis for sharing memory between host and VMs. On host, *XMemPod* is implemented and runs as a user space program, which manages host and remote shared memory block devices and disk block device, each device exposes a conventional block device I/O interface to the guest OS's VMA manager, which treats the block device as a fixed size swap partition. No modification is required to the host OS or any of its existing kernel modules, including KVM. The entire host shared memory area is divided into multiple elastic memory *pods*, one per VM on the host, provided that a shared memory pod is only established for a guest when paging initially occurs, and the shared memory pod is revoked from the guest when execution of its applications terminates, or when the host shared memory manager terminates its service to the guest VM. Upon creating a shared memory pod, its entire address space is logically partitioned into slabs of fix size, according to `Host-SlabSize` in initial configuration. The shared memory pod manager maintains the mapping between the page offset in the guest swap partition and the address in its corresponding shared memory swap partition, and it opportunistically adjusts the size of each shared memory pod to decrement or increment one slab, based on the lower and upper pod utilization thresholds, defined by `sm-pod-min` and `sm-pod-max`. There are three types of pages in the host shared memory: *active* pages, those being used, *free* pages, those allocated to the pod but not yet being used, and *idle* pages, those that do not belong to any pod.

The Remote Memory Manager (RMM) is designed with similar principles. First, the setting of `Remote-SlabSize`, `Remote-pod-min`, and `Remote-pod-max` can be different from those set for host shared memory configuration. Second, slabs are the units for balancing memory utilization across remote machines and providing low latency mapping to remote memory. All pages in a slab are mapped to the same remote memory. How-

ever, different slabs in the same remote pod can be mapped to multiple remote servers' memory. Third, we implement remote memory management on top of nbdX [22], which exploits the advantages of multi-queue implementation in the block layer and the Accelio acceleration facilities to provide fast IO to a remote shared memory device. When a paging request is redirected from a guest VM to the remote memory manager, it first selects one primary remote server and one secondary remote server from the `remote-count` candidates maintained by *XMemPod*, and uses the corresponding remote memory pods to expose the I/O interface of remote shared memory and use VirtIO to the guest (VM1) for read and write requests. For each remote server, a remote swap pod connector is setup to keep the RDMA connection with the remote server. On a remote node, when its NIC receives paging requests, it maps the target offset and size into the corresponding remote registered memory region, and uses DMA to access the pages. To guarantee the data to be delivered without corruption, RC QP (Reliable Connected Queue Pair) is configured on both sides. For page-in request (read), the SMM passes to RMM the metadata from the compressed disaggregated memory page table (CSPT), including compressed page size, location of the compressed page in the remote memory pod(s), the instruction on proactive page-in (batch read), RMM checks metadata parameters and performs page-in operations. RMM also broadcasts periodic resource utilization messages which the client modules use to discover the available remote memory servers, such as their memory availability and load as well as page transfers to its clients. When a sever reaches its remote memory capacity `Remote-pod-max`, it will decline to serve any new page-out (write) requests from its clients.

The Disk Swap Partition Manager (DSM) is called for serving swap-out requests in two situations: (i) When there is no shared memory available at host or the host shared memory partition reaches its hybrid swap-out threshold specified by `sm-pool-max`, and there are no available remote memory. (ii) When a remote page-out event occurs, *XMemPod* backs up each remote page-out event from local VM to its disk swap partition as well. This serves

as a fault tolerant contingency solution for network link failure or remote server failure.

4.4 Elastic Remote Memory Sharing

Demand-driven Memory Sharing. *XMemPod* creates a host-VM shared memory pod for each VM on a host and multiple remote memory pods for remote memory sharing by starting with unmapped state for each slab. *XMemPod* initiates host/remote placement to map the slab to a host-VM shared memory pod or a remote memory pod when additional external memory is needed. Such late binding reduces unnecessary slab mapping and improves memory sharing efficiency.

Connection Management. *XMemPod* uses one-sided RDMA READ/WRITE operations for data plane activities and RDMA SEND/RECEIVE operations for control plane activities. For each individual connection, two channels are established: (i) RDMA channel is used for maintaining the network connection, data transfer, and data corruption. (ii) *XMemPod* channel maintains status of remote MemPod by interacting with remote agent and uses it for placement and eviction algorithms.

Remote Memory Balancing. Several remote memory selection algorithms are implemented to minimize memory imbalance across machines and used as configuration parameters, including round robin (RR), weighted RR, random, or power of two choices [114, 21], with RR as the default. *XMemPod* proactively allocates memory slabs of size `Remote-memory-max` and registers as memory region for RDMA operations on remote servers, which reduces the cost of initialization process. Remote idle memory is monitored and when it drops below the `Remote-low` threshold, remote memory slabs will be deregistered preemptively through our remote slab eviction handler based on `Remote-SlabSize`, and updates the CSPT corresponding to the deregistered slabs. All future write requests will go to other remote servers through the remote memory server selection algorithm. All future read (page-in) requests will be directed to the secondary remote server if two or more remote servers are used to serve each remote paging-out request. If both remote servers

were no longer available due to remote slab eviction, the page-in requests will resort to the corresponding local disk.

Fault Tolerance. *XMemPod* runs its client module of the disaggregated memory paging service on a guest VM, and its server modules, such as SMM, RMM, DSM on the host in every node of the cluster. If the corresponding guest VM or the host machine fails, or the local disk fails during paging, such as server/VM crash, *XMemPod* provides the same failure semantics as the guest OS swap facility today. For remote memory paging, it is important to handle unexpected failure scenarios due to network connection (link) failure or remote server failures. First, *XMemPod* does not require central coordination for remote memory paging, thus, single point of failure and frequent message synchronization are avoided. Second, each remote swap-out operation is replicated to at least two remote nodes and also maintained in the local disk swap of the guest VM. Third, each remote paging operation is treated as an atomic transaction, all or nothing, which is recorded in the corresponding entry of the CSPT, thus removing the inconsistency due to remote connection failure or remote server failure. To support stronger fault tolerance, *XMemPod* supports paging to more than one remote servers by maintaining the metadata on the size of remote server available memory and sorting remote servers accordingly. This not only further increases the fault tolerance in the presence of network connection failure and unreachable server induced failure. It also simplifies the remote slab eviction handling.

Consistency. For each VM, we maintain a CSPT, which works as a log table to track of where a swap-out page is. We set the *CSPT_entry* to `LOCAL` if the page is stored in the host-VM shared memory pod, otherwise it is set to `REMOTE`. `REMOTE` tag has three location entries to record the address of the primary remote MemPod, the secondary remote MemPod, and the local disk partition corresponding to the VM. For page-out request, *XMemPod* synchronously writes data into RDMA channel of primary remote MemPod and secondary remote MemPod, while writing asynchronously to the local disk partition. Once the RDMA WRITE operation of two remote MemPod complete, the write operation is committed af-

ter its corresponding *CSPT_entry* is updated. To avoid data corruption during transferring process, RC QP is configured for each RDMA channel, which guarantes that messages are delivered from a requester to a responder at most once, in order and without correction. Requester considers a message operation complete once there is an ack from the responder that the message was read/written to its memory. Responder considers a message operation complete once the message was read/written to its memory. For page-in request, if the *CSPT_entry* is set to `REMOTE`, *XMemPod* will put one RDMA READ operation on the primary MemPod address into the RDMA dispatch queue. When the READ completes, *XMemPod* responds by marking the commit of the read operation. Otherwise, *XMemPod* reads it from the secondary MemPod, or the local disk partition. We consider two failure cases.

(1) **Local VM failure/exit**. When the PSM is unreachable, *XMemPod* considers the scenario as local VM failure or exit. Upon detecting this failure, the garbage collection is triggered at both local host-VM shared MemPod and two remote MemPods. It marks all slabs for this VM as unmapped and removes corresponding CSPT. The garbage collection for local disk is performed and cleared by guest OS.

(2) **Host failure**. For each server module of *XMemPod*, its RMM periodically checks the reachability of each remote MemPod agent in its remote server group, and considers unreachability of a remote agent as the remote host failure scenario. Upon detecting this failure, *XMemPod* updates the corresponding *CSPT_entry*(s) as unavailable. For a page-in read request, if both primary and secondary remote servers fail, then by its *CSPT_entry*, the request will be sent to the local disk swap partition. Also for each failed host, the remote agent unmaps and deregisters all slabs that are used as remote MemPod slabs for page requests read from and write to this failed host. The broken RDMA and *XMemPod* channel will be removed from the agent as well.

Scalability. A fundamental challenge for providing cluster-wide memory virtualization is to scaling the system to terabytes of collective memory in a cluster. However, to track

where each swap-out page is located in the cluster, the server module corresponding to the VM needs to maintain the metadata such as server ID, MemPod ID and offset in the CSPT entry of this page in the host-VM shared memory. Consider a simple in-memory hash table is used to store the location of each of its pages in the cluster, and each page is 4KB in size. If we use 8 bytes to store each location identifier metadata, then we would need up to 5GB host-VM shared memory to store the hash table for the 2 terabytes of cluster-wide memory. For 10 TB, it is 25 GB. Maintaining a CSPT of such size for each VM will incur prohibitively high cost as the cluster-wide memory scales up. To address this scalability challenge, *XMemPod* adopts group sharing model, where servers within the group forms a remote memory sharing group, each server can access the memory sharing state of all other members of its group. A leader election protocol [115] elects the one with maximum unused memory as the leader of a group periodically. If the leader node crashes (handshake time-out), a new leader election process will be triggered. Also, leaders of all remote memory sharing groups form the top tier grouping service, which supports dynamic re-grouping upon request from their member client(s).

XMemPod is deployed successfully on small size RDMA clusters (32 machines) with up to 8 VMs each, a total of up to 256 VMs. As the number of VMs increases on a host, there is small overhead involved in dynamic allocation/deallocation of shared memory pools in unit of slab. Right sizing of slabs is beneficial. Larger slabs can decrease the utilization efficiency of memory sharing. Also, the current sharing strategy for host and remote shared memory is greedy and demand driven.

4.5 *XMemPod* Optimizations

We describe three other optimizations. Figure 4.4 shows how they are orchestrated in concert for host/remote memory sharing.

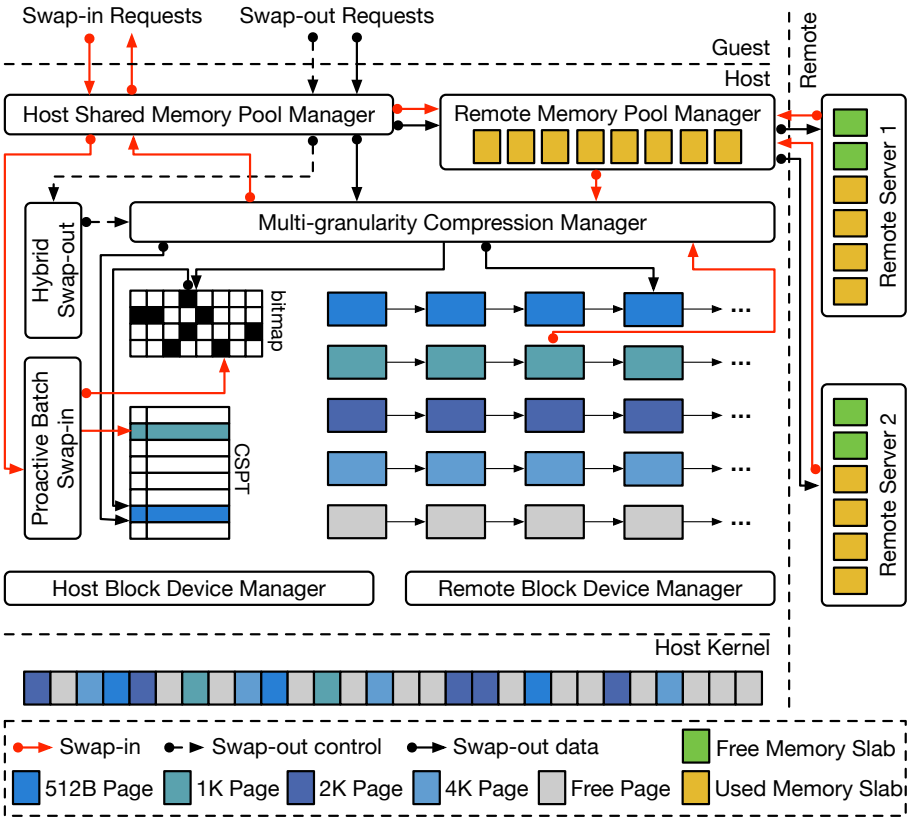


Figure 4.4: *XMemPod* Optimizations.

4.5.1 Compressed Swap Page Table

XMemPod compresses all swap-out pages before placing them into the proper swap partition. All swap-in pages are decompressed first before sent back to the application's working memory. Based on the notion that the compression efficacy mainly depends on two factors: compressibility of data and choice of data granularity [93, 94], *XMemPod* deploys a multi-granularity compression algorithm, which is optimized for achieving the best compressibility: e.g., a 4KB page can be best compressed to one of the four different sizes: (0,512B], (512,1KB], (1KB,2KB], and (2KB,4KB]. LZ4 [96] is used as the default. Other multi-granularity compression algorithms include LZO-1X [95].

To keep track of compressed swap pages, four compressed page placement queues are created: for 512B queue, 8 compressed page slots are created for one shared memory page, each holds a compressed page of size in (0,512B]. Similarly, 4 compressed page slots for

1KB queue, each slot holds a compressed page of size in (512B, 1KB], 2 slots for 2KB queue in (1KB,2KB], and 1 slot for 4KB queue in (2KB,4KB]. *XMemPod* creates and maintains a compressed disaggregated memory page table (CSPT) to keep track of each swap-out page and its metadata With CSPT, the swap-in operation can easily lookup and locate the compressed page from the respective swap partition, and send it back to the guest in decompressed format. A hash-table based index structure [116, 117] is used to optimize read/write operations on the CSPT. When a swap-out operation is intercepted, a *CSPT_entry* is created for the swapped out page and links it to the Hashtable and a bitmap variable field is used to indicate the location of the swap partition where the compressed page currently resides. Other metadata recorded in the *CSPT_entry* includes the address of this compressed page, the length of compressed page which is required for decompression.

4.5.2 Hybrid Page-out

Upon receiving a page-out request from the guest VMA, the PSM intercepts the page and calls the *swap_writepage()* operation. The operation begins by compressing the page into a temporary buffer in order to know which one of the four compressed page-groups this compressed page should belong to. Based on the compressed page size, *XMemPod* chooses the shared memory page in the most suitable compressed page group, locates the free slot with its offset, and has temporary buffer copied into the corresponding slot address. Next, the PSM checks with SMM to determine if there is sufficient room in the host-VM shared memory pod to serve this page-out request. When unused host memory is insufficient, instead of putting the **most recent pages to remote** (MRPR), which keeping the older pages in the fast host-VM shared memory pod, we advocates the hybrid page-out optimization by placing **least recent pages to remote** (LRPR), which maximizes the utilization of host-VM shared memory to serve the most recent paging traffic, and selects remote memory as the next preferred tier for hybrid swap-out. Similarly, when there is insufficient remote memory in the cluster, the hybrid page-out procedure will split the older pages by their LRU stamps

and move the relatively more recent pages to remote memory and the older ones to disk swap partition. In both cases, pages remain compressed during hybrid swap-out process and updates are performed on the compressed disaggregated memory page table and the hashtable index accordingly.

4.5.3 Proactive Page-in

Acceleration Through Metadata Piggybacks. We observe that when the working set of application is big, the page table grows and the PTE lookup cost increases. For frequent swap-in events, this cost can be non-trivial. *XMemPod* accelerates swap-in operations by piggyback of some metadata: when a page is swapped out, *XMemPod* keeps the address of the PTE related to this page as a piece of metadata together with the swapped-out page, and stores the metadata in the corresponding entry of the compressed swap page table CSPT for this swap-out page. For each page of 4K bytes, this metadata only takes up 4 bytes. The cost of keeping this metadata in the swap area is only about 1/1000 of the total size of the swapped-out pages. Using this metadata, when a page is swapped into the working memory of the guest, *XMemPod* is able to quickly locate the PTE that needs to be updated by referring to this metadata without the need to scan the page table. Thus, the time spent on accessing the PTE of a swapped page will not increase as the size of the system wide page table grows. This advantage is applied to proactive swap-in from host to guest, from remote to guest, or from remote to host.

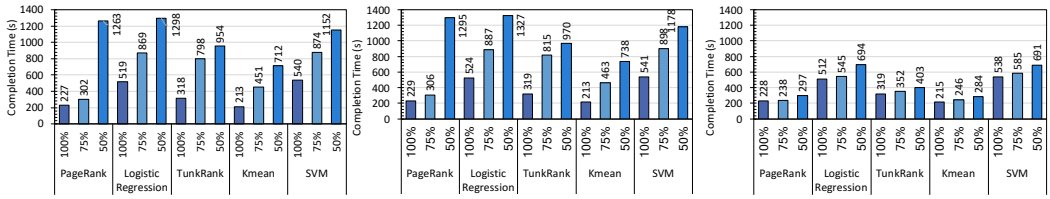
Proactive Batch Read (swap-in). If a page is swapped in from either remote memory pod(s) or a disk swap partition, the SSM will check if the free pages in the shared memory pod are above the pre-defined batch swap-in threshold, supplied in system initialization: if yes, the remote memory pod manager or the disk swap block device will trigger proactive batch swap-in, which batch swap-in those pages nearby the requested page (temporal or spatial locality aware), and otherwise, it only swaps in the requested page.

Table 4.1: Applications Used in Experiments

Workload	Suite/Application	Dataset
PageRank	Spark GraphX	1 million pages
LogisticRegression	Spark Mllib	7.5 million samples
TunkRank	PowerGraph	30 million vertices
Kmean	PowerGraph	7 million samples
SVM	Liblinear	45 thousand samples
YCSB-Memcached	Memcached	20 million records
YCSB-Redis	Redis	20 million records
YCSB-VoltDB	VoltDB	20 million records
Canneal	PARSEC	2.5 million records
Apache Solr	Cloudsuite	12GB index

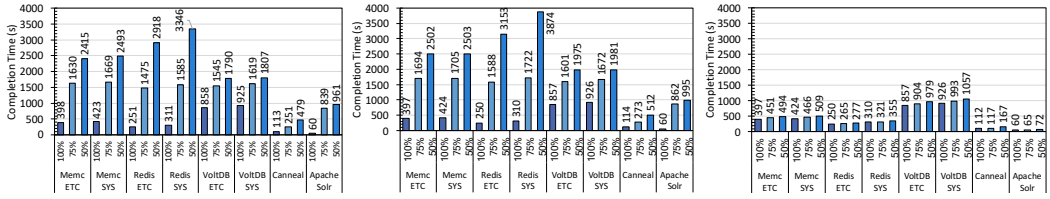
4.6 Evaluation

We evaluate *XMemPod* using ten popular memory-intensive applications, with first five machine learning and next five big data applications, listed in Table 4.1, and compare *XMemPod* with Linux (conventional OS swap facility), Accelio nbdX [22] and Infiniswap [21]. Experiments are performed on a 32-machine, 56 Gbps Infiniband cluster. Each machine has 32 core E5-2650v2 CPU, 64 GB memory, 2TB SATA 7.2K rpm hard drives, and running KVM 1.2.0 with QEMU 2.0.0 as virtualization platform. We use Linux 4.1.0 and Ubuntu 14.04 for both the guest and host system. For most of the experiments unless otherwise stated, we run 80 VMs on a 32-machine RDMA cluster and created an equal number of VMs for each application workload. 1/3 of VMs used 100% configuration, 1/3 of VMs used 75% configuration, and 1/3 of VMs used 50% configuration. We started with the 100% configuration by creating a VM with large enough memory to fit entire workload in memory. We measured the peak memory consumption, and then ran 75% and 50% configurations by creating VMs with enough memory to fit these fractions of the peak memory consumption. The working sets for the ten applications range from 25GB to 30GB and their input dataset sizes range from 12GB to 20GB. Unless explicitly stated, we use the default configurations: `sm-SlabSize=512MB`, `Remote-SlabSize=1GB`, hybrid swap-out threshold 80%, proactive swap-in threshold 50%, and remote server connection timeout is `1s`.



(a) nbdX (b) Infiniswap (c) XMemPod

Figure 4.5: Machine Learning Workload Performance Comparison of Infiniswap, nbdX and *XMemPod*.

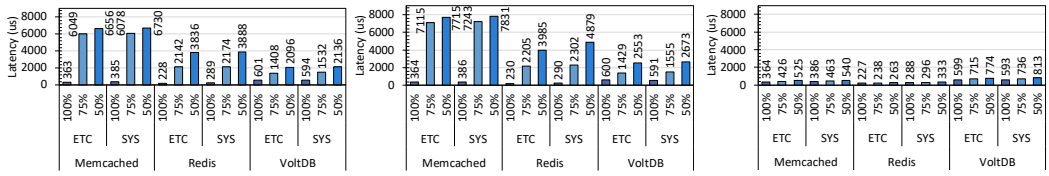


(a) nbdX (b) Infiniswap (c) XMemPod

Figure 4.6: Big Data Workload Performance Comparison of Infiniswap, nbdX and *XMemPod*.

4.6.1 Impact on Applications

Machine Learning Applications Performance. This experiment measures the *XMemPod* performance (Figure 4.5c) on PageRank, LogisticRegression, TunkRank, Kmean, and SVM by comparing them to those using Infiniswap (Figure 4.5b), nbdX (Figure 4.5a) and Linux. Due to space limit, Linux measurement figures are omitted, but we provide statistical results in the discussion. We highlight three observations: First, for 75% configuration, *XMemPod* improves application completion time by 24x on average and up to 83x, over Linux, improves over Infiniswap and nbdX by 2.3x and 2.2x respectively. Second, for 50% configuration, *XMemPod* improves application completion time by 45x on average and up to 85x over Linux, improves over Infiniswap by 4.4x (best case) and 2.6x on average, improves over nbdX by 4.3x (best case) and 2.5x on average. Third, both nbdX (Figure 4.5a) and Infiniswap (Figure 4.5b) show a super-linear increase of completion time, whereas *XMemPod* (Figure 4.5c) shows steady performance. For 75% configuration, using *XMemPod*, machine learning applications experience only on average 1.1x increase in



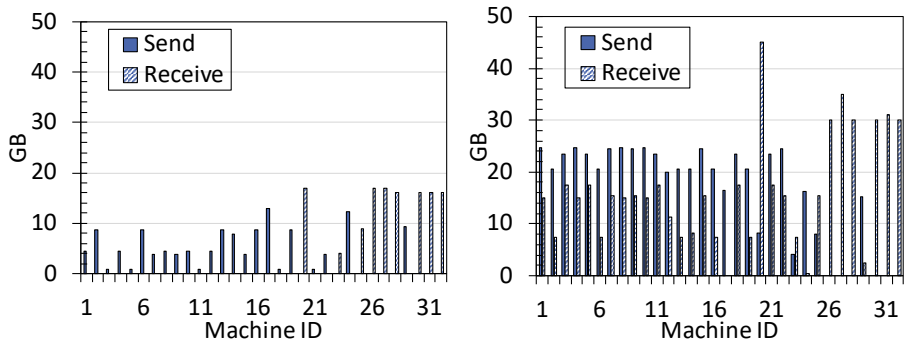
(a) nbdX (b) Infiniswap (c) XMemPod

Figure 4.7: Big Data Workload 99th Percentile Latency Comparison of Infiniswap, nbdX and *XMemPod*.

completion time instead of 1.9x using Infiniswap, 1.8x using nbdX and 25x using Linux. For 50% configuration, using *XMemPod*, machine learning applications experience only on average 1.3x increase in completion time instead of 3.4x using Infiniswap, 2.5x using nbdX and 59x using Linux.

Big Data Applications Performance. Applications used in this set of experiments are Memcached [1], Redis [2], VoltDB [3], Canneal [118], and Apache solr [119]. We measure *XMemPod* performance (Figure 4.6c) and compare it to Infiniswap (Figure 4.6b), nbdX (Figure 4.6a) and Linux. For Memcached, Redis and VoltDB, we use the dataset published by Facebook [120] and choose ETC and SYS as workloads to explore different rates of SET operations. ETC is read-intensive workload, which has 5% SET and 95% GET, and SYS has 25% SET and 75% GET, which is popularly used as write-intensive workload [120, 21]. The experiments start by populating a Memcached or Redis or VoltDB by YCSB client, as shown in Table 4.1. We then perform ETC and SYS workload over the whole dataset.

We observe three interesting results: (1) For 75% configuration, *XMemPod* improves application completion time by 138x on average and 423x in the best case, over Linux, improves over Infiniswap and nbdX by 13.2x and 12.9x in the best case, and 4.7x and 4.5x on average, respectively. (2) For 50% configuration, *XMemPod* improves application completion time by 612x in the best case and 271x on average over Linux; It improves application completion time by 13.8x and 13.3x in the best case and 6.6x and 6.2x on average over Infiniswap and nbdX respectively. (3) This set of experiments shows that applications using *XMemPod* benefit from more steady performance for both types of big data work-



(a) XMemPod Send/Receive Data (b) Infiniswap Send/Receive Data

Figure 4.8: Network Traffic of Individual Machines.

loads, but when using Infiniswap or nbdX, they experience high degree of fluctuations in performance.

For 50% configuration, using *XMemPod*, big data applications experience on average 1.2x increase in completion time, compared to 7.9x using Infiniswap, 7.3x using nbdX and 313x using Linux.

Given that tail-latency is considered a more accurate indicator of performance degradation for bigdata workloads, we measured median and 99th percentile response latency for Memcached, Redis, and VoltDB respectively by using all three paging systems. Due to space limit, we only include the 99th percentile latency comparison of *XMemPod* with Infiniswap and nbdX in Figure 4.7 since application performance using Linux is significant slower for all workloads. Using *XMemPod*, the 99th percentile latencies for all applications are much closer to the performance of 100% configuration when their working sets do not fully fit in memory. For example, for 50% configuration, using *XMemPod*, the 99th tail latencies of Memcached, Redis, and VoltDB are increased by 42%, 16%, and 33% respectively, and in contrast, the 99th tail latencies are increased by 19.7x, 16.0x, and 3.4x respectively using Infiniswap, and by 16.9x, 13.9x, and 2.5x respectively using nbdX.

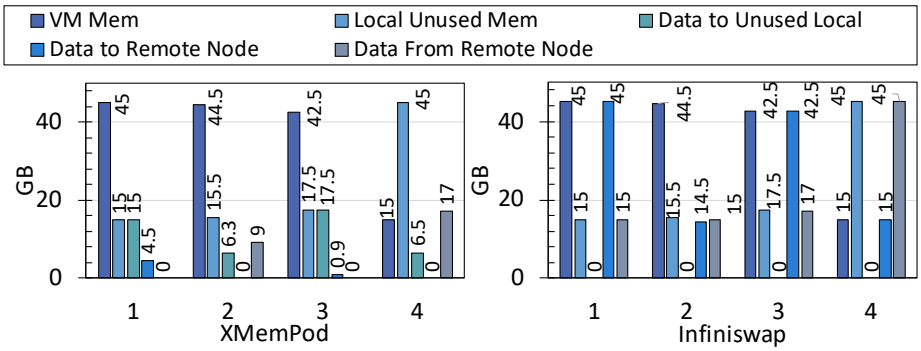
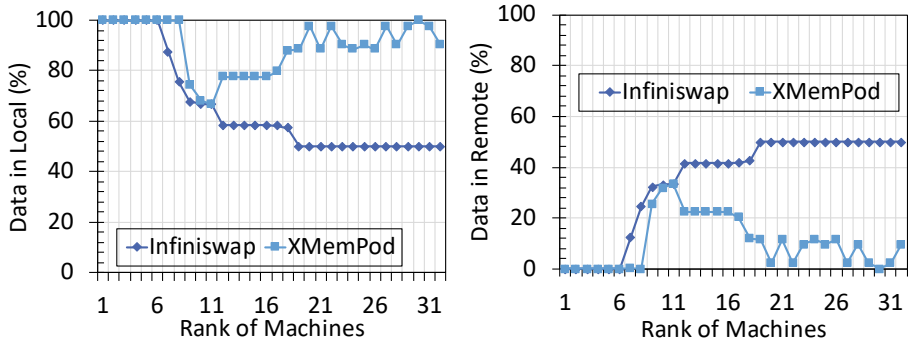


Figure 4.9: Memory Usage Distribution.



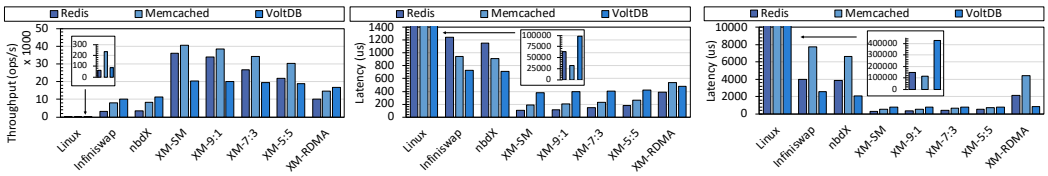
(a) Data Stored in Local (%) (b) Data Stored in Remote (%)

Figure 4.10: Data Distribution of Individual Machines

4.6.2 Cluster Utilization

Figure 4.8 shows the amount of send/receive data over the RDMA cluster of 32 machines. The total amount of network traffic over RDMA in the case of Infiniswap is 524GB, while *XMemPod* is 128GB, which is only 24% of Infiniswap. Similar comparison is observed for nbdX.

Figure 4.9 shows the detailed measurement on memory usage distribution of four randomly picked machines for *XMemPod* and Infiniswap respectively. For Infiniswap, all servers do not benefit from unused local host memory. Although servers 2 and 4 have enough local unused memory to serve the external memory service request from its local VMs, Infiniswap is unable to do so, and instead, all page-out requests are sent to other remote nodes. For *XMemPod* server 1 utilizes both local host and remote host memory



(a) Throughput (b) Median Latency (c) 99th Latency

Figure 4.11: Varying Host and Remote Memory Sharing Distribution (different latency scale in Figure 4.11b and 4.11c).

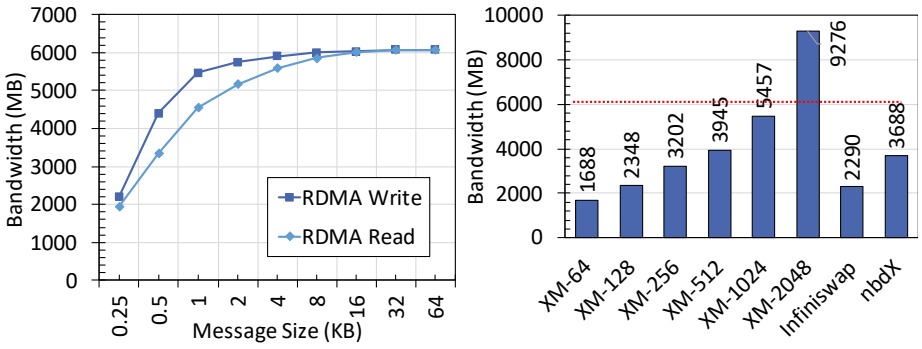
to serve the paging requests from local VMs. Servers 2, 3 and 4 can serve all page-out (write) requests from unused host memory pods, in addition to provide remote disaggregated memory services to other nodes in the cluster.

Figure 4.10 compares the overall data distribution with respect to data stored in local host v.s. data stored in remote nodes for *XMemPod* and Infiniswap. It is interesting to observe that for Infiniswap, the ratio of the average of data stored in local and remote is 65% v.s. 35%. In contrast, for *XMemPod*, the ratio of average of data stored in local and remote is 90% v.s. 10%.

4.6.3 Host/Remote Memory Distribution

This section compares the performance impact of varying host and remote memory distribution on application performance for all four systems: Linux, Infiniswap, nbdX, and *XMemPod*. Using 50% configuration, we have 50% of working set sent to external disaggregated memory via paging. For *XMemPod*, XM-SM denotes 100% of paging events are handled in host shared memory. XM-RDMA denotes 100% in remote memory via RDMA network, and zero host shared memory is reserved by *XMemPod*. The remaining three host-remote distribution split ratios are XM-9:1, XM-7:3, and XM-5:5. XM-9:1 denotes that 90% of paging traffic is served in host-VM shared memory pod and 10% of it is sent to the remote memory pods.

Figure 4.11 shows the results. We highlight four observations. First, Figure 4.11a shows that using XM-SM, throughputs of Redis, Memcached, and VoltDB increase by up to 571x,



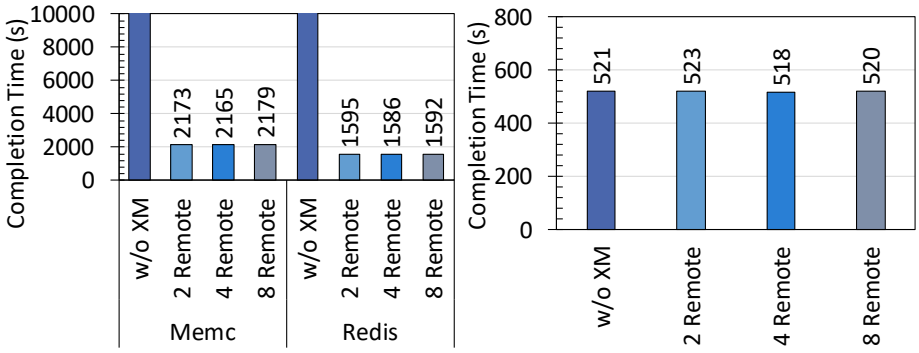
(a) RDMA Bandwidth (b) Software Stack Bandwidth

Figure 4.12: Bandwidth Measurement.

171x, and 240x respectively, compared with Linux, increase by 11.4x, 5.1x, and 2.0x compared with Infiniswap, and increase by 10.5x, 4.9x, and 1.8x compared with nbdX. Second, using XM-RDMA, throughput of Redis, Memcached and VoltDB increase by 3.2x, 1.8x, and 1.6x respectively, compared to Infiniswap, and increase by 2.9x, 1.8x, and 1.5x respectively, compared to nbdX. Third, in *XMemPod*, as the percentage of remote memory increases in the hybrid swap-out setting, ranging from XM-SM, XM-9:1, XM-7:3, XM-5:5 to XM-RDMA, throughputs of all three applications reduce linearly in Figure 4.11a. Finally, using *XMemPod*, the mean and 99th percentile latencies for Redis, Memcached, and VoltDB are significantly shorter compared to Linux, Infiniswap, and nbdX for all settings of host-remote memory paging, as shown in Figure 4.11b and Figure 4.11c. Even with XM-RDMA, the 99th percentile latencies of Redis, Memcached and VoltDB improve by 1.8x, 1.7x, and 3.1x, comparing to Infiniswap, and by 1.7x, 1.5x, and 2.6x, comparing to nbdX.

4.6.4 Memory Bandwidth Utilization

To better understand the reason why *XMemPod* RDMA only scenario (XM-RDMA) outperforms Infiniswap and nbdX, we compare the RDMA network bandwidth and software stack bandwidth of *XMemPod*, Infiniswap and nbdX. Figure 4.12 shows the results. The bandwidth of RDMA write/read operation is measured using various message sizes, rang-



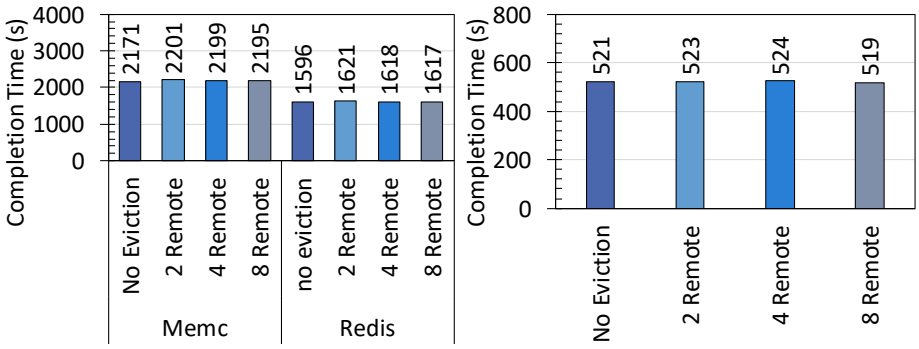
(a) Host - Memcached and Redis (b) Remote - Logistic Regression

Figure 4.13: *XMemPod* Performance with Multiple Remote Servers

ing from 1/4KB to 64KB. Message size indicates how much each RDMA write/read operation sends/receives at one time. The RDMA network card is Mellanox FDR CX3 single port mezz card with 56Gbps. Figure 4.12a shows that bandwidth reaches maximum as message size is equal or greater than 16KB. The max bandwidth is around 6GB, which matches the speed of RDMA network card (56Gbps). Figure 4.12b measured the software stack bandwidth, i.e., how fast (*XMemPod*, Infiniswap, or nbdX) could get data from block device and put it to RDMA driver. As the message size increases, the bandwidth also increases (XM-64 means XMemPod with 64KB message size). When message size reaches 2048KB, the software stack bandwidth is over the RDMA bandwidth, which is highlighted with red dotted line. Overflowed requests would be put into RDMA work request send queue, which will occupy extra memory. Therefore, to avoid overflow, request sending rate of *XMemPod* should be bounded by the RDMA bandwidth. Hence, 1024KB is used as the default for *XMemPod* when using RDMA network card with 56Gbps capacity.

4.6.5 Impact of Remote Memory Sharing

In this section, we deploy *XMemPod* on a 32-machine RDMA cluster on Emulab [110]. We turn off the host-VM shared memory option and only rely on remote memory for serving paging requests under 50% configuration. We randomly choose a server with two VMs running Memcached and Redis ETC workload respectively, and configure their remote



(a) Host - Memcached and Redis (b) Remote - Logistic Regression

Figure 4.14: Eviction Impact on Performance

memory slabs to be distributed across multiple remote servers. We measure the performance of remote paging to 2 to 8 servers, each of which runs a VM for Logistic Regression workload. We report two sets of experiments due to space constraint.

Remote Paging to Multiple Servers. Figure 4.13 measures the impact of remote paging on both host and remote application performance. We make three observations. *First*, on host server, using *XMemPod*, all applications complete significantly faster than the scenario without *XMemPod*. Furthermore, *XMemPod* with 2 remote server case improves the completion time of Memcached and Redis by 70x and 179x respectively, over the scenario without *XMemPod*. *Second*, the performance of both Redis and Memcached is stable as we increase the number of servers for serving remote paging. This is because each server only equips with one Infiniband card in the Emulab RDMA cluster, the performance of remote read/write operation is limited by one network card. We expect that *XMemPod* will benefit from parallel remote I/O if multiple Infiniband cards are equipped. *Third*, Logistic Regression on each of the remote servers serving remote paging traffic from Redis and Memcached exhibits stable performance in all cases, which is expected because *XMemPod* uses one-sided RDMA operation to read/write remotely, which completely bypasses remote CPUs.

Impact of Eviction on Application Performance. Figure 4.14 measures the impact of remote memory eviction operations on the performance of both host and remote appli-

Table 4.2: Hybrid Swap-out Policy Comparison: Completion Time (second)

Application	Method	XM-9:1	XM-7:3	XM-5:5
LR	**LRPR**	701(22%)	714(25%)	728(24%)
	MRPR	903	947	961
SVM	**LRPR**	593(19%)	607(22%)	621(21%)
	MRPR	729	776	787

cations. We highlight two observations. First, eviction increases the completion time of Memcached and Redis by 1.3% and 1.4%. A primary factor is caused by the process of `EVICT` event handling, updating CSPT (section 4.5.1), and redirecting request to the other remote server that stores the copy of requested memory slab. Second, eviction has no performance impact on Logistic Regression on remote server, as shown in Figure 4.14b. The main cost of evicting slab is memory region deregistration. For 1GB slab, the average cost of it is 273 microseconds. When we increase to 4GB, 8GB or 16GB of slabs, the average cost for memory region deregistration is 1121, 2157, and 4356 microseconds respectively.

4.6.6 Effect of Optimization

Due to the space constraint, we report the experimental evaluation on the effectiveness of multi-granular compression, hybrid page-out and proactive page-in provided in *XMemPod* in this Appendix for those readers who are interested in the results.

Effect of Hybrid Page-out. We compare the Most Recent Pages to Remote (MRPR) and the Least Recent Pages to Remote (LRPR) with three hybrid page-out settings (XM-9:1, XM-7:3, and XM-5:5). For all 10 applications, hybrid swap-out with LRPR policy performs much better than MRPR. Table 4.2 shows the results for Logistic Regression and SVM due to space limit. By LRPR, most recently accessed pages are more likely to be swapped out to the host shared memory swap space, whereas the older pages are moved to the remote memory swap partition or disk swap. Thus, LRPR is more efficient in utilizing the smaller but faster host memory when it is available, whereas using MRPR, most recently accessed pages are in the remote memory swap space, resulting in longer average access time.

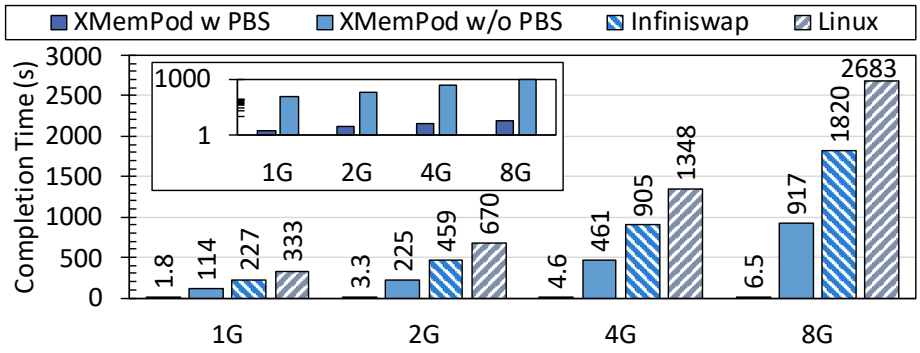
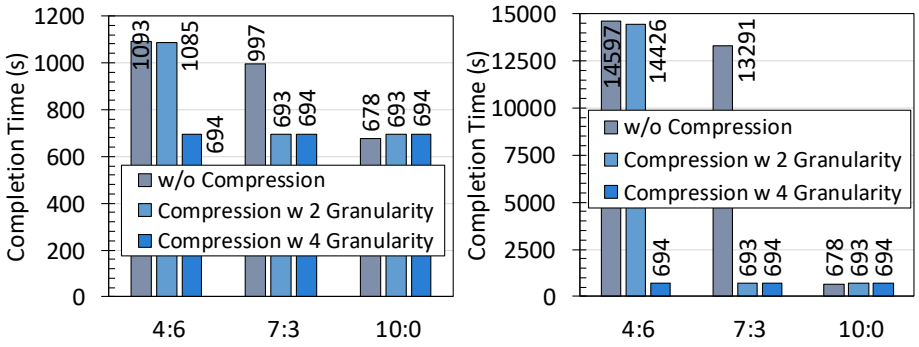


Figure 4.15: Proactive Batch Swap-in (PBS) (zoom-in figure shows the comparison of *XMemPod* w and w/o PBS)

Effect of Proactive Batch Swap-in (PBS). We measure the total time spent on paging-in different amount of memory pages under four scenarios: *XMemPod* with PBS, *XMemPod* without PBS, Linux and Infiniswap. Figure 4.15 shows the results. We make two observations: (1) Although as the total amount of memory pages to be swapped in is increasing, the total time spent on swap-in increases for all four scenarios. *XMemPod* with PBS is significant faster than Infiniswap and Linux. It takes 1820 seconds and 2683s to swap in 8GB memory in Infiniswap and Linux case respectively. In comparison, with proactive batch swap-in, the total time consumed by the same swap-in events is only 6.5s, which is three orders of magnitude smaller than Infiniswap and Linux. (2) When the page-in amount is increased from 1GB to 8GB, using Infiniswap, the completion time is increased by 1,593 seconds (from 227s to 1820s). In comparison, using *XMemPod* with PBS, as the amount of swap-in memory pages grows, the completion time only increases by 1.8 seconds to 6.5 seconds.

Effect of Multi-granular Compression. We evaluate the impact of multi-granular disaggregated memory page compression on the completion time of applications, under 50% configuration, for three scenarios: (i) 4-granularity compression (*XMemPod* with 4 compressed page sizes), (ii) 2-granularity compression (*XMemPod* with 2 compressed page sizes), and (iii) no compression scenario. Similar results are observed from all applications and we report the measurements on Logistic Regression (LR) due to space limit. We first



(a) Remote Shared Memory (b) Local Disk

Figure 4.16: Effect of Multi-granularity Compression

obtain the peak size of 28GB to fit the LR working sets in memory for the 100% configuration. Then for 50% configuration, 14GB of LR working set cannot fit in memory. We vary the ratio of paging to host shared memory and to remote memory by 4:6, 3:7 and 10:0 respectively. With 4:6 ratio, about 40% of the 14GB LR working set can leverage from the host-VM shared memory pod and the remaining 60% will use the remote memory. We measure the swap-out performance with and without compression. Figure 4.16 shows that multi-granularity compression improves the performance of applications as more swap-out pages can remain in the host-VM shared memory. We highlight two observations. First, with 4:6 host-remote memory ratio, Figure 4.16a and Figure 4.16b show that *XMemPod* with 4-granularity compression improves the performance of Logistic Regression by 1.57x and 21x for hybrid swap-out to remote memory or to local disk respectively, compared to *XMemPod* w/o compression scenario. But *XMemPod* with 2-granularity compression cannot keep most of the swap-out pages in the host-VM shared memory, in comparison. Second, with 7:3 host-remote memory ratio, without compression, the performance of Logistic Regression is 1.57x and 21x slower respectively for hybrid swap-out to remote memory or to local disk, compared to *XMemPod* with either 2-granularity compression and 4-granularity compression, because most of the swap-out pages under both multi-granularity compressions can be served from the host-VM shared memory.

4.7 Related Work

Dynamic Memory Balancing. Existing research on scheduling and consolidating physical memory among VMs centered on dynamic memory balancing, with Ballooning [31] as the most representative. Ballooning requires manual administration of when to trigger ballooning and how much ballooning is sufficient. Proposals [32, 33, 34] devoted to periodic estimation of VM working set size. But, accurate working set prediction is hard under changing workloads [8, 35, 36].

Memory Page Compression and Deduplication. Existing proposals for page deduplication [7, 121, 122] aim at identifying duplicate memory pages, reducing the amount of unnecessary or inconsistent paging, to provide better memory sharing among VMs. Some open source efforts provide transcendent memory (tmem) interface for paging [123] or a compressed RAM cache [99] via mounting a ramdisk for VM paging. However, these solutions are not scalable to increased paging requests and suffering high cost. Proposals for addressing double paging [91] and memory compression [124, 125, 126, 127, 128, 129, 130, 94] can be leveraged in *XMemPod* to further improve host/remote memory sharing.

Distributed Shared Memory/Disaggregated Memory. Distributed shared memory (DSM) was studied extensively [37, 38, 39, 19, 40, 41]. However, DSM suffers poor performance due to high communication overhead. Disaggregated memory has attracted much attention recentlly [42, 43, 17, 44, 15, 45]. Most proposals rely on new hardware architecture, new network protocols to cut down the communication cost. Some proposals show the benefit of leveraging RDMA technology [46, 47, 48, 49, 50, 51, 52, 48], exploiting the disk-network latency gap by limiting remote memory to certain workloads, such as remote storage for key-value stores[53, 54, 55, 56], distributed objects [57], swap pages [58, 59, 60, 61, 62, 21], object replication [63]. Most of these efforts lack of desired transparency and all existing proposals treat and leverage unused host memory as the remote memory, fail to take advantage of the performance gap between DRAM and network interconnect, such as

Infiniband. Until the network interconnect offers the same level of low latency I/O as the DRAM, we argue that a latency-sensitive disaggregated memory orchestration framework is critical for efficient virtualization of cluster wide memory.

4.8 Conclusion

Virtualization of cluster wide memory via disaggregated memory orchestration is a key to enable independent scaling of compute and memory resources, allowing VMs to meet transient memory demand to run large memory applications in virtualized clouds. We have presented *XMemPod*, a software defined, application transparent disaggregated memory orchestration framework for virtualization of cluster-wide memory. *XMemPod* is deployed on a virtualized RDMA cluster without any modifications to user applications and the OSes. It offers orders of magnitude performance gain for bigdata and ML applications. *XMemPod* incurs no functional nor performance impact on guest VMs that do not use it or that incurs zero paging. Our extensive experimental evaluation of *XMemPod* on unmodified Spark, Apache Hadoop, Memcached, Redis and VoltDB have demonstrated its effectiveness: Compared to conventional OS disk swap, *XMemPod* improves throughputs of these applications by 11x to 612x, median and tail latencies by 174x to 702x and 218x to 630x respectively. Compared to the state of art remote memory paging systems represented by Infiniswap and nbdX, *XMemPod* improves throughputs by up to 14x, median latencies and tail latencies by up to 12x and 15x respectively.

CHAPTER 5

SUPPORTING HUGE PAGES FOR BIG DATA AND ML APPLICATIONS

We describe *Xpage*, a huge page memory management framework. *Xpage* by design can effectively mitigate the fragmentation problem and brings the benefits of huge page to applications automatically and transparently, even in high frequent memory allocation and deallocation condition which can quickly fragment memory space and degrade application performance significantly. Instead of demoting huge pages to base pages once fragmentation occurs [131, 132, 133], *Xpage* tracks allocated/unallocated memory regions and compacts all memory regions in memory allocator level by placing them carefully within huge page boundaries. *Xpage* represents a memory management redesign that brings performance and memory saving to memory intensive applications by supporting dynamic huge page memory management without resorting to splitting huge pages for memory fragmentation.

5.1 Introduction

A fundamental property of virtual memory is that the CPU references a virtual address that is translated via a combination of software and hardware to a physical address. This allows data only to be paged into memory on demand improving memory utilization. The use of virtual memory is pervasive but this indirection is not without cost. Due to translation, a virtual memory reference necessitates multiple accesses to physical memory, multiplying the cost of an ordinary memory reference by a factor depending on the page table format. To cut the costs associated with translation, virtual memory implementations take advantage of the principal of locality by storing recent translations in a cache called the Translation lookaside buffer (TLB). It is a part of memory management unit (MMU). The TLB stores the recent translations of virtual memory to physical memory and serves as an address

translation cache. Once an instruction asks the processor to do memory operation on a virtual address, the processor first checks to see whether the TLB contains an entry for that virtual address. If the requested address is present in the TLB, the search yields a match quickly and the retrieved physical address can be used to access memory. This is called a TLB hit. If the requested address is not in the TLB, it is a miss, and the translation proceeds by looking up the page table in a process called a page walk. The page walk is expensive when compared to the processor speed, as it involves reading the contents of multiple memory locations and using them to compute the physical address. Modern servers (such as 64 bit computers) have the capacity to support hundreds of gigabytes (GBs) of RAM, and memory intensive applications should be able to take advantage of such large amount of DRAM memories. On the other hand, increased capacity also poses a significant challenge for address translation.

Hardware manufactures and operating systems provide kernel support for larger page sizes in terms of 2MB huge pages instead of traditional 4KB pages to address the increasing DRAM capacity and the increased demand for efficient access to large memory. The benefits of huge pages are the obvious performance gain from fewer translations requiring fewer cycles. It has been reported that database workloads will gain about 2-7% performance using huge pages whereas scientific workloads can range between 1% and 45% [134].

Challenges of supporting huge pages. Transparent Huge Page (THP) [131, 132] and libhugetlbfs (a userspace interface to utilize huge pages) [135] are used to enable a single TLB entry to map a large region (2MB). Huge pages require size aligned addresses and the memory allocator only allocate huge pages if mmap region is 2MB aligned. But applications do not fully utilize the huge pages allocated to them, because a process using less than a full huge page has to reserve the entire huge page region, which could introduce serious internal fragmentation. Such greedy and aggressive allocation of huge pages makes it impossible to predict application's memory usage because of the memory fragmentation.

Moreover, memory intensive applications such as MongoDB, Redis and VoltDB running with huge pages enabled servers often experience performance degradation, high kernel space CPU utilization, memory bloating, even for memory intensive applications [133]. As a result, to maintain predictable and stable performance for memory intensive applications, many strongly recommend users to disable huge page even though they are running memory intensive applications on large memory machines.

Scope and Contributions. We describe a huge page memory management framework, called *Xpage*. *Xpage* by design can effectively mitigate the fragmentation problem and brings the benefits of huge page to applications automatically and transparently, even in stressful conditions. For example, high frequent memory allocation and deallocation can quickly fragment memory space and degrade application performance significantly, *Xpage* always keeps huge pages compacted and achieves the expected performance benefits of huge page. Concretely, instead of demoting huge pages to base pages once fragmentation occurs [131, 132, 133], *Xpage* tracks allocated/unallocated memory regions and compacts all memory regions in memory allocator level by placing them carefully within huge page boundaries. *Xpage* represents a memory management redesign that brings performance and memory saving to memory intensive applications by supporting dynamic huge page memory management without resorting to splitting huge pages for memory fragmentation.

5.2 Linux Huge Page

Operating systems can leverage the hardware support for huge pages in two ways. First, reserving a pool of contiguous memory at boot time. Linux provides a `libhugetlbfs` interface to utilize huge pages. But, this approach is not flexible and is not transparent for applications, since it requires applications to explicitly request for huge pages. Pages that are used as huge pages are reserved inside the kernel and cannot be used for other purposes. Huge pages cannot be swapped out under memory pressure. Once a number of huge pages have been pre-allocated to the kernel huge page pool, a user with appropriate privilege

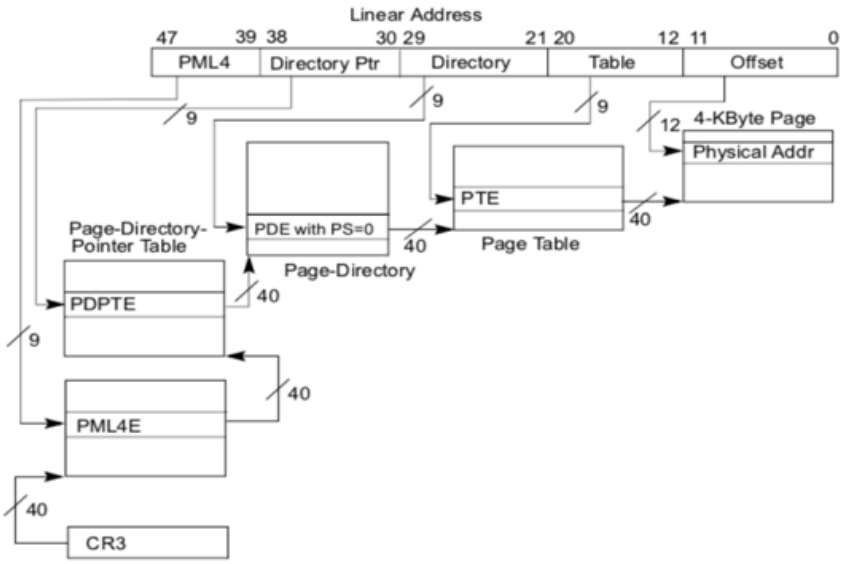


Figure 5.1: Linear Address Translation to 4K Page. [136]

can use either the `mmap` system call or shared memory system calls to use the huge page. Windows and MacOS provide similar approach to support huge pages. Second, a more practical way to bring the benefits of huge pages to applications is Transparent Huge Pages (THP), where operating system tries to use huge pages for the backing of virtual memory with huge pages that supports the automatic promotion and demotion of page sizes and without the shortcomings of `libhugetlbfs`. Linux huge page management algorithms are aggressive. It always tries to allocate huge page if THP is enabled. Linux's approach works well for simple applications that allocate a large memory region and rarely release objects, but in most cases applications do not always fully utilize the huge pages allocated to them, especially when application does frequent memory allocation and deallocation, which leads to serious internal fragmentation problems.

For huge page promotion, Linux page fault handler adds all memory area `mm_slot` into a single global list to be used by `khugepaged`, which is a kernel daemon that occasionally attempts to allocate a huge page. If it succeeds, it will scan through that list looking for a place where that huge page can be substituted for a bunch of smaller pages. The scanning process is costly since it requires to scan page tables and checks page status, including page

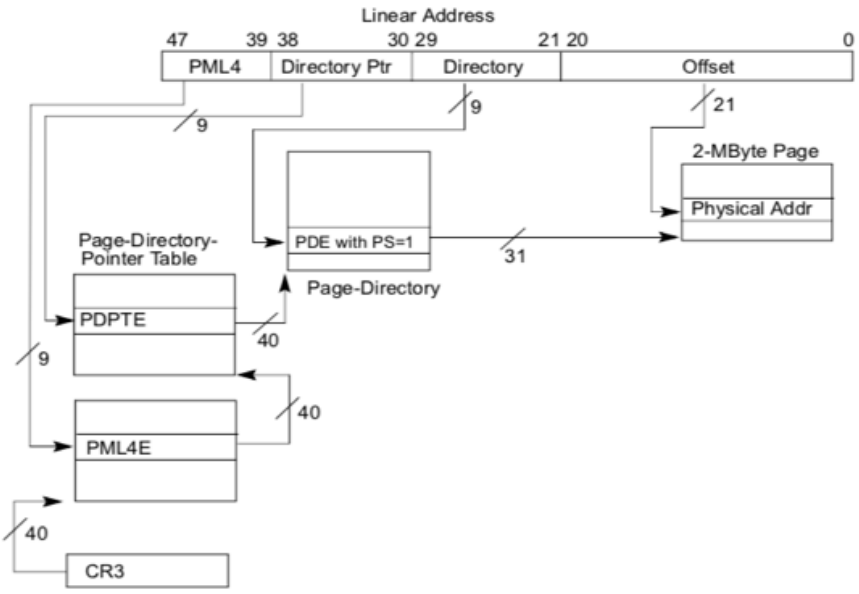


Figure 5.2: Linear Address Translation to 2M Page.[136]

LRU, page lock, access bit, anonymous bit, idle page flag, and reference bit. The complete promotion process includes four steps. 1) lock page table 2) scan to find candidates 3) allocate huge page 4) copy base page data to huge page region. `collapse_huge_page()` is the API for triggering huge page demotion in Linux.

For huge page demotion, once a huge page partially unmapped it will be put onto split queue. Pages on split queue are going to be split instantly or under memory pressure according to different configuration. Demotion is also expensive since this action implies splitting all PMD the page mapped with. For instance, we load 10 million key-value pairs into Redis and randomly remove 50% of whole dataset. The average latency of splitting one huge page is 40,964 nanosecond. `split_huge_page_to_list()` is the API for triggering huge page promotion in Linux.

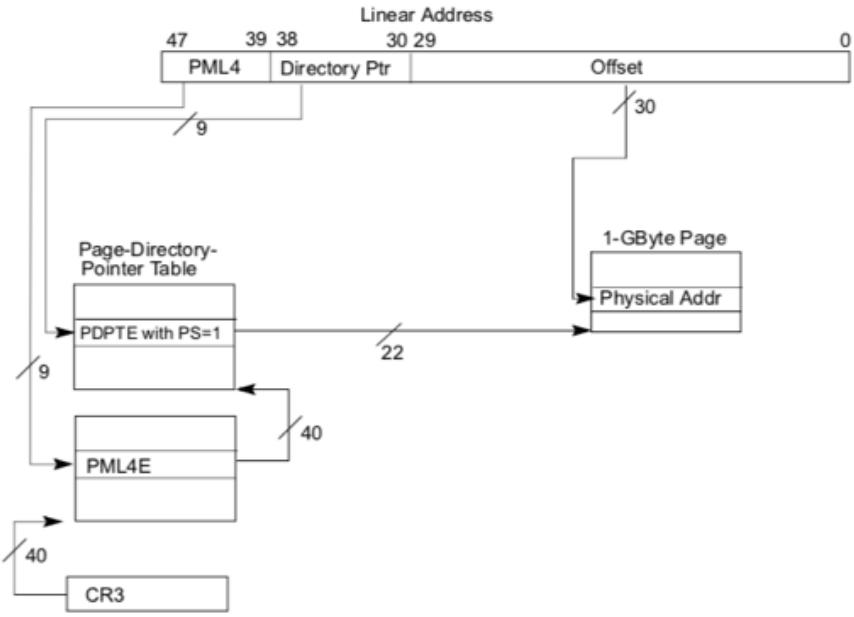
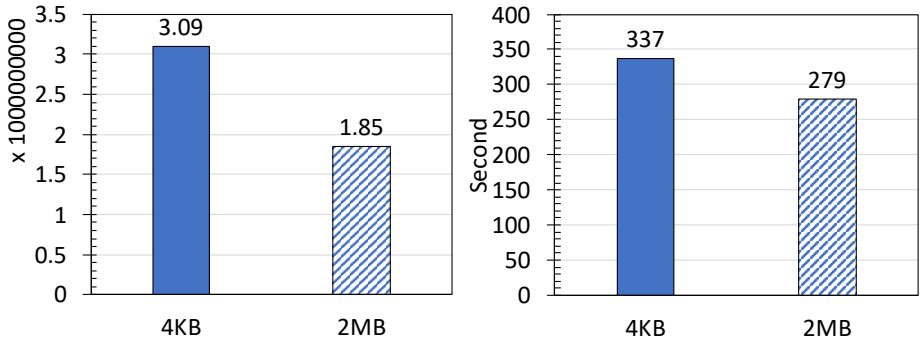


Figure 5.3: Linear Address Translation to 1G Page.[136]

5.3 Problem Statements

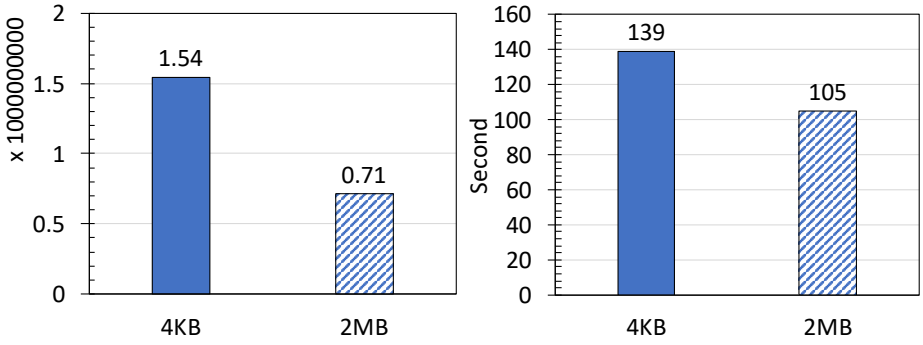
5.3.1 Paging Structure

Currently, hardware manufactures support three different paging modes, which are 32-bit, PAE, and 4-level paging. Among them, only 4-level paging mode can be adopted for huge pages, since it supports 4KB, 2MB, and 1GB page size, as shown in Figure 5.1, Figure 5.2, and Figure 5.3. With 4-level paging, linear address are translated using a hierarchy of in-memory paging structures located using the contents of CR3. A 4KB naturally aligned PML4 table is located at the physical address specified in CR3. A PML4 table comprises 512 64-bit entries (PML4Es). A PML4E is selected and combined with PDPT (Page Directory Pointer Table) to select a PDPTE. Similarly, PDE (Page Directory Pointer Table Entry) and PTE (Page Table Entry) are selected and are combined with offset to eventually generate physical address. In each level of paging structure, a `PS` bit is used for indicating page size, which is the key for huge pages support. If `PS` bit is 0, the entry references another level of paging structure, while the `PS` bit is 1, the entry maps a physical page. Using



(a) Number of TLB Miss (b) Completion Time

Figure 5.4: Redis Performance with Load Workload



(a) Number of TLB Miss (b) Completion Time

Figure 5.5: Redis Performance with Read Workload

`PS` bit, operating system could figure out what the page size is. Specifically, 4KB page uses the last 12 bits as offset; 2MB page uses PTE (9 bits) and last 12 bits as offset; 1GB page uses PDE (9 bits), PTE (9 bits), and last 12 bits as offset. By splitting or collapsing paging structure, operating system could dynamically promote base pages to huge pages or demote huge pages to base pages. This will be addressed in section 5.3.5.

5.3.2 Performance Benefit from Huge Pages

One huge page is composed of a fixed number of contiguous base pages. Linux uses a 2MB page entry to cover a contiguous 2MB memory region for its address translation, instead of using 512 4KB page entries to cover it. With huge page, the page table becomes significant smaller. Since PTE (9 bits) pluses the last 12 bits are used as offset, each huge page could

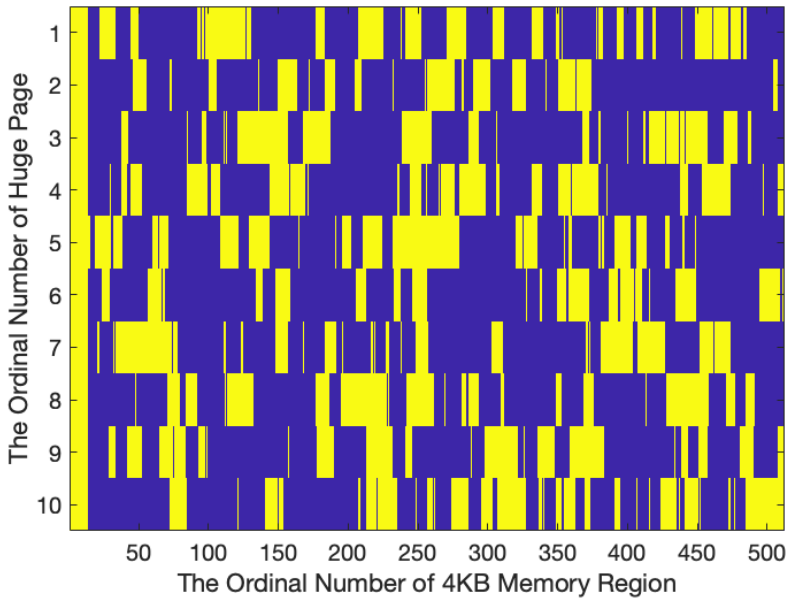


Figure 5.6: Huge Page Internal Fragmentation.

save 32KB (512 * 64Byte) memory space compared with using base page. Moreover, TLB table of the same size could cover significant larger memory space.

To gain an in-depth understanding of the potential opportunities of exploiting huge pages, we use YCSB to generate 10 million key-value pairs and execute write and read operations to Redis using different page size, as shown in Figure 5.4 and Figure 5.5. We highlight two observations. First, compared to 4KB page size, using 2MB page size the the number of TLB miss is reduced by 40% and 54% for write and read workload respectively. Second, compared to 4KB page size, using 2MB page size the workload completion time is reduced by 17% and 24% for write and read workload respectively.

5.3.3 Internal Fragmentation

Internal fragmentation is caused by applications do not fully utilize the huge page allocated to them. Due to the rules governing memory allocation, computer memory is pre-allocated through memory allocator using `mmap` or `brk` system calls. Once THP is enabled, memory allocator always gets a huge page size (2MB) memory chunk from operating system kernel.

Table 5.1: Memory Utilization of Huge Pages

Huge Page	Occupied (4KB)	Unoccupied (4KB)
1	192	320
2	129	383
3	168	344
4	185	327
5	191	321
6	147	365
7	168	344
8	185	327
9	181	331
10	187	325

Bins are used by memory allocator to store free memory regions. Each bin has an associated size class and stores/manages memory regions of this size class. According to bin configuration, memory allocator divides memory chunk into smaller memory regions and assigns them to related bins. Basically one huge page memory space contains many memory regions that belongs to different bins and all memory allocation requests are served in bin level. Only when the whole memory chunk is empty does the memory allocator returns the mapped huge page back to OS kernel. In other words, as long as one memory region with one huge page boundary is still occupied by a object, the entire huge page is reserved. Figure 5.6 shows the internal fragmentation problem when running Redis. We populate 10 million key-value pairs with 1KB objects using YCSB and delete 50% of the keys randomly. We sample the memory layout of 10 huge pages, as shown in Figure 5.6. Each row is one 2MB huge page, which is divided into 512 4KB memory regions. We use two colors to indicate the status of each 4KB memory region, yellow means in use, blue means free. Even though the average utilization of 10 sampled huge page are only 34%, none of them could be released back to OS kernel because of the internal fragmentation. The detailed memory utilization of each huge page is shown in Table 5.1. The problem could be magnified when workload contains large amount of delete operation. It is impossible to predict an application's memory usage in production because memory usage depends on huge page usage, which in turn depends on the fragmentation. Unlike other types of

fragmentation, internal fragmentation is extremely difficult to reclaim. Existing approach includes THP and Ingens, which are discussed in section 5.3.5.

5.3.4 Copy-on-Write

Copy-on-write (COW) is a scheme by which a memory range that is duplicated into a new region is not physically allocated a new page frame until some data in one of the two pages is actually modified. In general there are two approaches to have COW work with huge pages. First, copy-on-write with huge pages. At first, the original huge page frame remains, and the two virtual huge pages each point to it. Both are marked read-only so that any attempts to write the huge page will trap into OS. When write happens, the huge page frame gets duplicated, and each virtual huge page is reassigned to point to its own copy of the huge page. The problem of this approach is that once the original huge page is internally fragmented, copy-on-write would also produce another internally fragmented huge page, making memory fragmentation problem even worse . Second, splitting huge pages to base pages, freeing unoccupied base pages, and then performing COW on base page granularity. Although this approach saves more memory space and has little change of current Linux implementation, accessing the split large pages could significantly degrades access performance and increase TLB miss ratio.

5.3.5 Existing Solutions and Limitations

One alternative of THP [131, 132] is Ingens [133], which is a framework for huge page support implemented in Linux kernel space. Ingens allocates only bases pages in the page fault handler and tracks each base page allocations. If the utilization with a huge page region is enough (90% in their implementation), Ingens promotes these base pages to a huge page asynchronously. Similarly, when a base page is freed within a huge page, Ingens updates the utilization of that huge page region. When the utilization drops below a threshold, Ingens demotes the huge page and frees the base pages that are released. How-

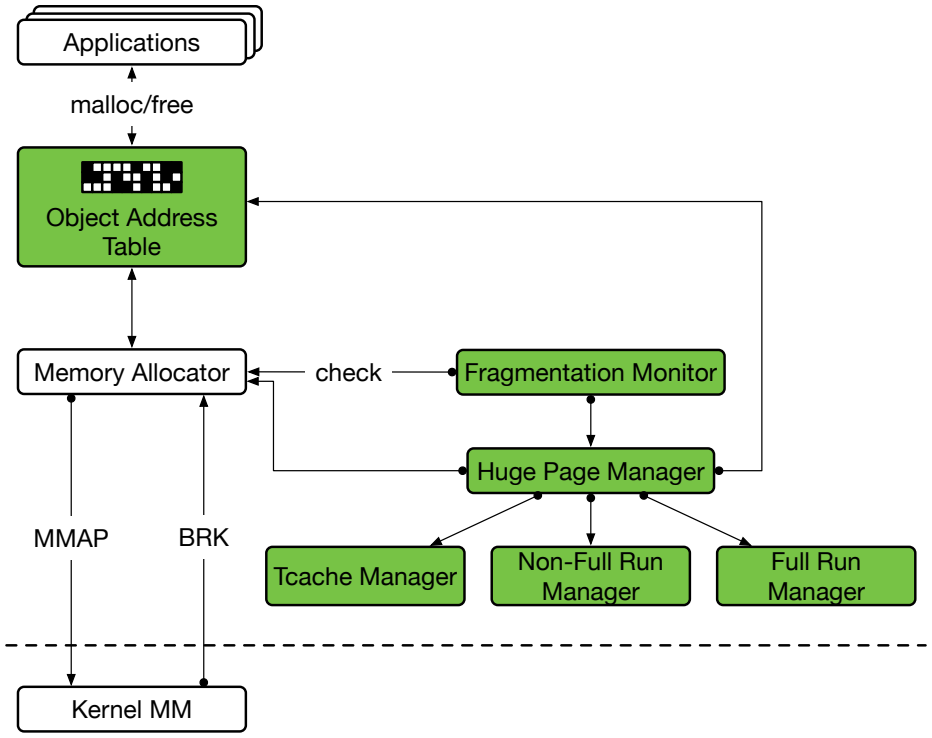


Figure 5.7: *Xpage* Overview.

ever, Ingens suffers from same problem as Linux and even worse. 1) Async promotion introduces around 50% performance drop, which will be addressed in §5.5 2) Access bit track required by utilization monitor adds additional cost on critical memory read/write path. 3) Using demotion to deal with memory fragmentation loses the benefits of using of huge page. After splitting huge pages, the memory space covered by TLB can be significantly reduced. Even though fragmentation is reduced, accessing the split large pages significantly increases TLB miss ratio and degrades access performance. 4) Both Linux and Ingens don't directly promote base pages in place within huge page boundary. In contrast, they have to copy base pages to a pre-allocated huge page, introducing the cost of memory copying in kernel space.

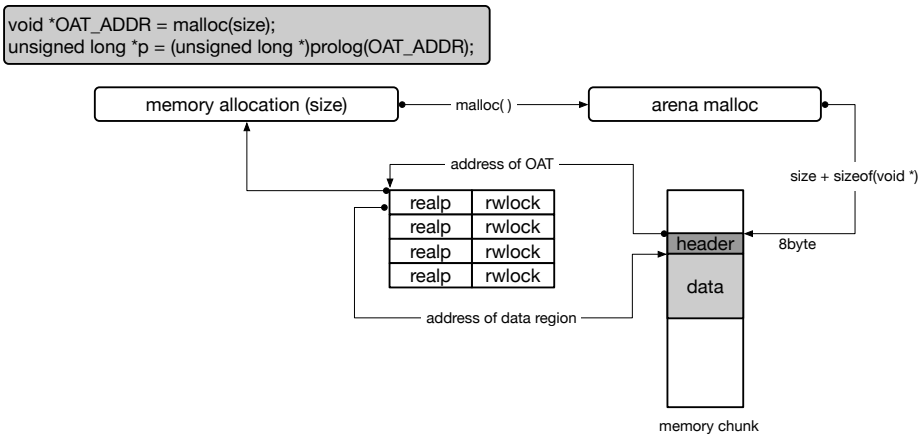


Figure 5.8: Memory Allocation Workflow.

5.4 *Xpage* Design

5.4.1 Overview

Xpage consists of three core components: the object address table (OAT), the fragmentation monitor (FM), and the huge page manager (HPM), which interacts with three sub managers that are tcache manager (TM), non-full run manager (NRM), and full run manager (FRM), as shown in Figure 5.7. They coordinate closely to bring high performance and efficient memory saving to memory intensive applications with dynamic memory behavior. Unlike THP and Ingens that are running in kernel space, *Xpage* is a memory allocator level memory management framework. We have the following considerations as designing *Xpage*. 1) The basic unit of memory management in OS kernel is the page, regardless base page or huge page. So, kernel could only handle external fragmentation but not internal fragmentation, since it has no knowledge of exact memory usage inside one page. This is why THP and Ingens could only sacrifice performance for reducing fragmentation by demoting huge pages. In contrast memory allocator interacts with both application (`malloc/free`) and kernel (`brk/mmap`), it has full picture of memory layout, which is required for dealing internal fragmentation problem. 2) To ensure fairness and security, we need to deal with internal fragmentation within each process's memory space and use each process's CPU

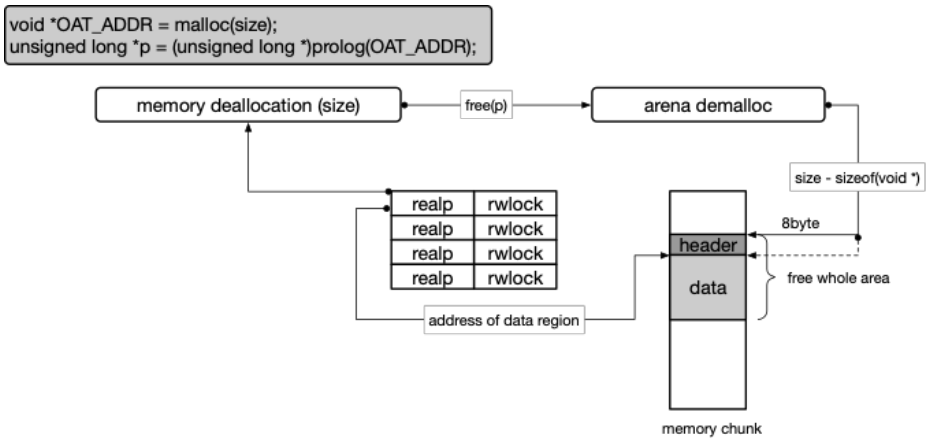


Figure 5.9: Memory Deallocation Workflow.

time to execute tasks. Memory allocator is ideal because it is a run-time library and has no permission to access other process's memory space. 3) Kernel has to manage and schedule kernel level threads as well as processes. It requires a full thread control block to maintain information about the thread. As a result there is significant overhead and increased in kernel complexity. Especially, we expect high frequency of memory allocation and deallocation requests, so running *Xpage* as run-time library should be more efficient than a kernel thread.

Huge Page Manager (HPM) intercepts all memory allocation and deallocation request. Memory space are managed in a hierarchical way. One huge page (2MB) is a memory chunk. A chunk is broken into several runs. Each run is actually a set of one or more contiguous pages. Each run holds regions of a specific size. Region applies to the end user memory areas returned by *Xpage*. For each arena bin, *Xpage* creates two pairing heap (PH), which provides O(log n) time for delete and delete_min and O(1) for all other operations, to track the start addresses of fully utilized runs (`fullrun_heap`) and non-fully utilized runs (`nonfullrun_heap`). `nonfullrun_heap` is used to choose the non-full run that is lowest in memory when memory allocation request occurs. This tends to keep objects packed well in low memory space and also help reduce the number of not fully utilized memory chunks. Fragmentation Monitor (FM) records the number of memory regions that

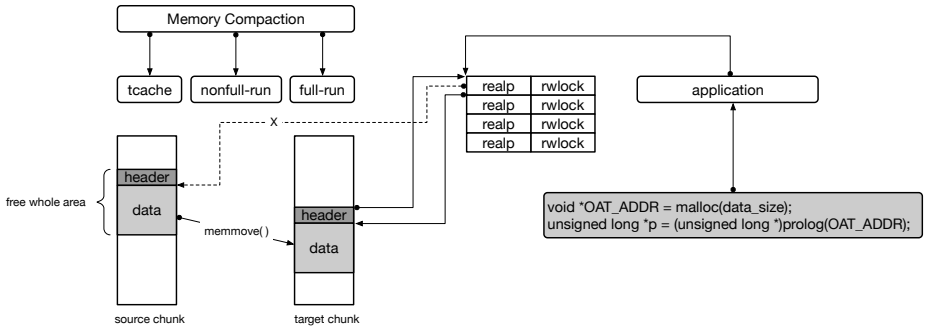


Figure 5.10: Memory Compaction Workflow.

are used as well as the number of memory chunks that are allocated. Internal fragmentation can be calculate by using the size of unused memory regions divided the size of all memory chunks. Once the internal fragmentation is over the threshold (80%), Fragmentation Monitor interacts with Huge Page Manager to trigger memory compaction, which contains tcache, non-full run, and fullrun compaction. The detailed compaction workflow is discussed in section 5.4.4. Object Address Table (OAT) is used to make applications access objects transparently after compaction. OAT is another level of indirection between application and memory allocator. Instead of a real address, *Xpage* returns application object a indirect pointer, which points to a entry in the OAT and can be parsed to the real address where the object data is actually stored. *Xpage* is responsible for maintaining this table and always updating it with the latest address once it moves data.

5.4.2 Memory Allocation Workflow

Upon receiving a memory allocation request, the *Xpage* HPM intercepts the `malloc()` and calls the `arena malloc` to ask requested size plus 8Byte (the size of a void pointer) free memory region from memory allocator. At the same time, HPM searches for a free entry in the OAT and use the first 8Byte of the requested memory region to store the entry address. Each entry in the OAT contains two items, the real pointer (`realp`) which points to the memory region that actually stores the object data, and the rwlock which is a read/write lock for guaranteeing the consistency of address. Lastly, *Xpage* returns the ad-

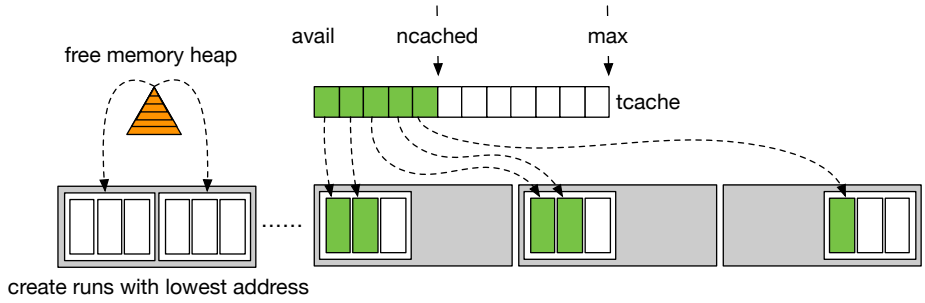
dress of OAT entry back to the application as a void pointer. The whole process of memory allocation is shown in Figure 5.8. On the application side, *Xpage* provides a `prolog()` function to check the rwlock first and then use the `realp` to access the actual memory region. Application side operations can be easily combined into compiler to make it transparent.

5.4.3 Memory Deallocation Workflow

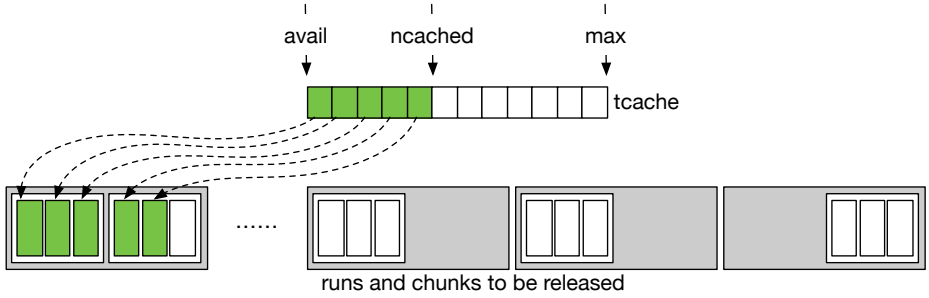
Upon receiving a memory deallocation request. *Xpage* HPM intercepts the `free()` and calls the `arena demalloc` to release memory region specified by the pointer `p`, as shown in the Figure 5.9. Since the pointer is pointing to the real memory region (data area in Figure 5.9) and the 8Byte before this pointer address (header area in Figure 5.9) stores the entry address in the OAT, these both memory spaces should be released. This process includes two steps. First, *Xpage* gets the OAT entry address using the pointer that is stored in the header and reclaims this entry memory space for future use. Second, *Xpage* releases the whole header and data area back to run. If no region exists in current run, *Xpage* releases this run. If no run exists in current chunk, *Xpage* release the chunk as well.

5.4.4 Memory Compaction Workflow

Upon the internal fragmentation is over the threshold, Fragmentation Monitor notifies the Huge Page Manger to execute memory compaction in tache, non-fullrun, and fullrun level. **Tcache Compaction**. Tcache is a caching system in memory allocator that shortens the memory allocation function call path. It is a shortcut pointer that directly points to the memory region, saving the cost of going through the hierarchical memory storage (chunk, run, and region) to search for available memory region. To compact Tcache, *Xpage* uses `free_memory_heap` to create empty runs in the lowest memory space, divide the runs into memory regions according to the bin size, and migrates each Tcache slot pointer to the new created memory regions, as shown in Figure 5.11. After compaction, if the run is



(a) Before Compaction



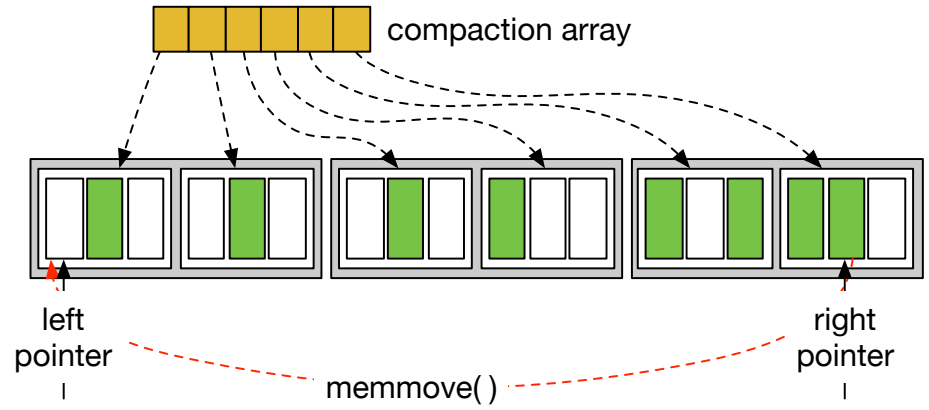
(b) After Compaction

Figure 5.11: Tcache Compaction.

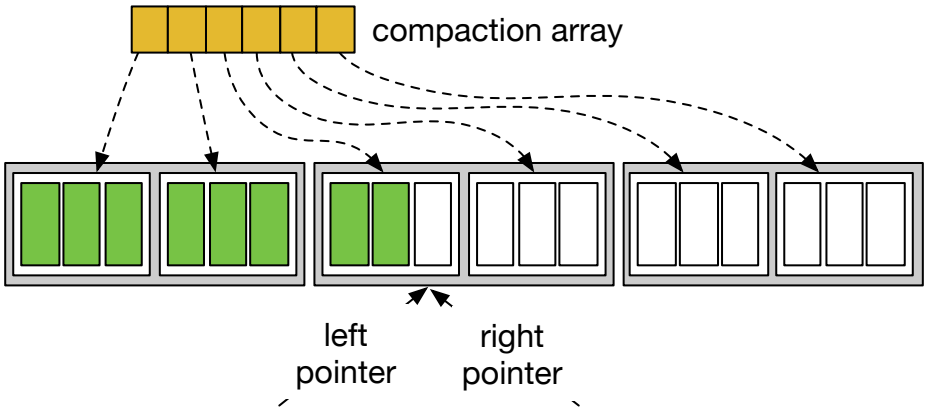
empty, *Xpage* will release it and merge with adjacent empty runs if possible. If the memory chunk is empty, *Xpage* will release the chunk. Or if the run is partially utilized, *Xpage* will insert it into `nonfullrun_heap`, which will be handled in the non-fullrun compaction process.

Non-fullrun Compaction. This process aims to reduce the number of not fully utilized runs. Firstly *Xpage* exports addresses of all non-fullrun from `nonfullrun_heap` to a sorted compaction array. *Xpage* runs as two separate algorithms. The first of the them starts at the beginning of the compaction array and searches for the free memory region which could be used as the target of migration. Meanwhile, at the end of the compaction array, the other half of the algorithm is search for occupied memory region which could be moved. Once *Xpage* finds the pair of free and occupied memory regions, it moves data by using `memmove()`. Eventually the two algorithms will meet somewhere toward the middle of the compaction array and non-fullrun compaction is finished.

Fullrun Compaction. This process aims to pack all data from high address to low address.



(a) Before Compaction

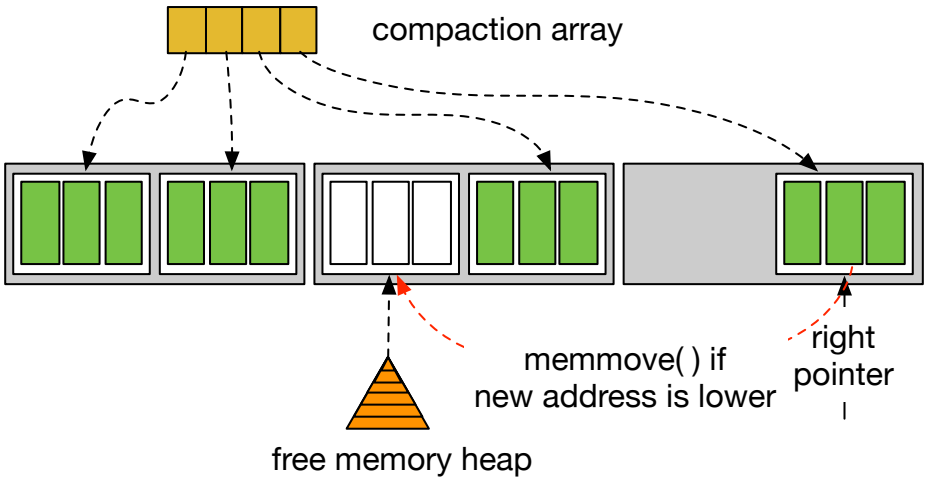


(b) After Compaction

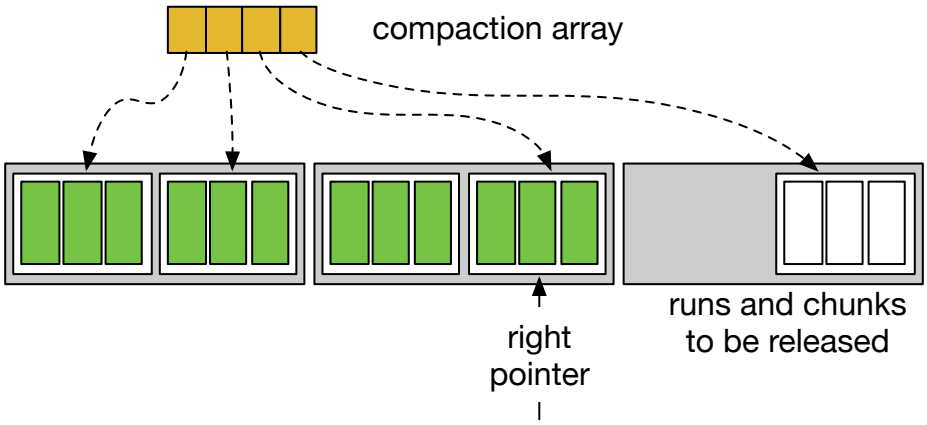
Figure 5.12: Non-fullrun Compaction.

Similarly, *Xpage* exports addresses of all full run from `fullrun_heap` to a sorted compaction array and use `free_memory_heap` to create an empty run in the lowest memory space. Meanwhile, it uses a pointer to scan each address from the end of the compaction array to the left. If the pointer address is higher than the address of the new created run, it means memory space is still fragmented, *Xpage* will migrate the run pointed by the pointer to the new created run. Or if the pointer address is lower, it means all runs are packed well already and Fullrun compaction is finished.

During compaction process, when memory region requires to be migrated, *Xpage* first locks the OAT entry and copy data to the new memory region using `memmove()`. Then, it reads the OAT entry address using the header and updates the `realp` as the new memory region address. Lastly, *Xpage* writes the OAT entry address to the new memory region



(a) Before Compaction



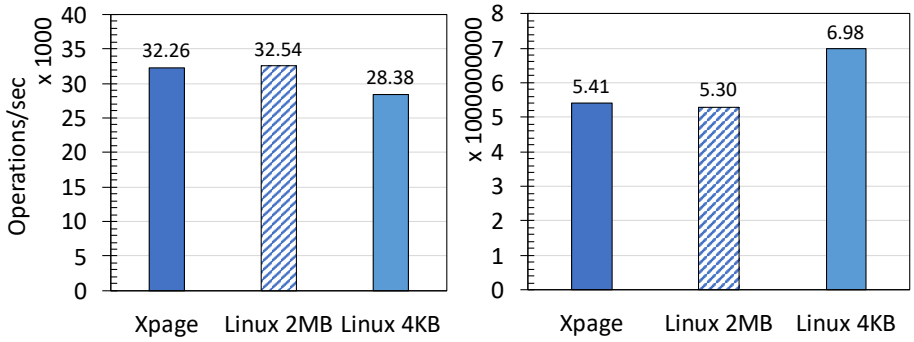
(b) After Compaction

Figure 5.13: Fullrun Compaction.

header area and releases the rwlock, as shown in Figure 5.10.

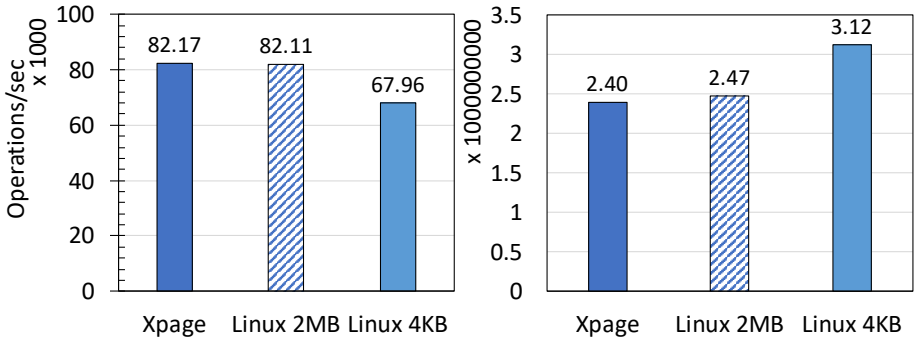
5.5 Evaluation

All experiments are performed on the server which has 32 core E5-2650v2 CPU, 64GB memory, 2TB SATA 7.2K rpm hard drives. We use Linux 4.1.0 and Ubuntu 14.04 as operating system. Intel support multiple hardware page sizes of 4KB, 2MB, and 1GB. Our experiments use only 4KB as base pages and 2MB as huge pages. Our experiment compares *Xpage* with Linux with Jemalloc 4.5 and Ingens with commit version fb1f78e for Redis 4.0 using workload generated by YCSB 0.14.0.



(a) Throughput (b) Number of TLB Miss

Figure 5.14: Load Operating Performance



(a) Throughput (b) Number of TLB Miss

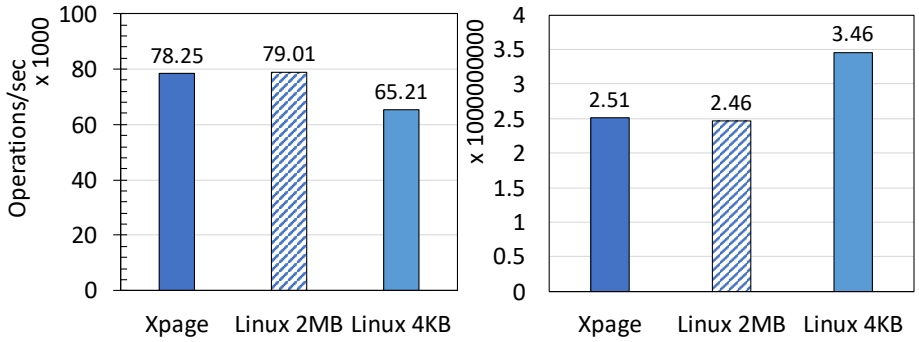
Figure 5.15: 50% Write and 50% Read Performance

5.5.1 General Performance and Overhead

We first measure and compare the performance of *Xpage*, Linux with 2MB huge page, and Linux with 4KB base page, as shown in Figure 5.14 for load workload, Figure 5.15 for 50% write and 50% read workload, Figure 5.16 for 25% write and 75% read workload, and Figure 5.17 for 5% write and 95% read workload. The dataset is generated by YCSB with 10 million key-value pairs. We highlight several observations. First, for all four experiments, *Xpage* improves the application throughput by 1.19X on average and up

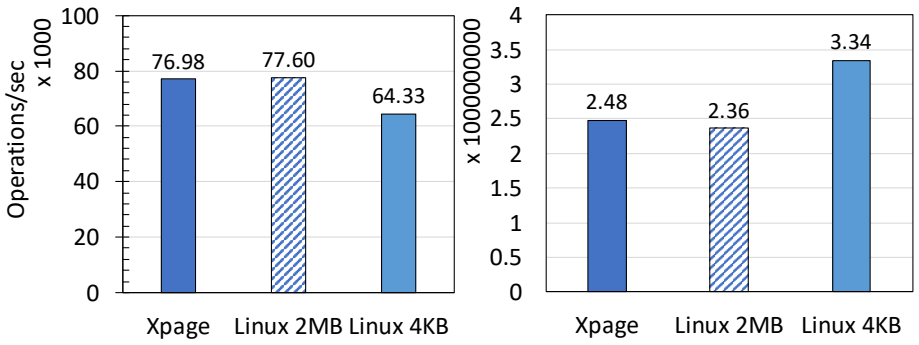
Table 5.2: Memory Access Latency.

	Memory Access Latency
Linux 4KB Base Page	88783330 ns
Linux 2MB Huge Page	54722308 ns
Xpage	54974789 ns



(a) Throughput (b) Number of TLB Miss

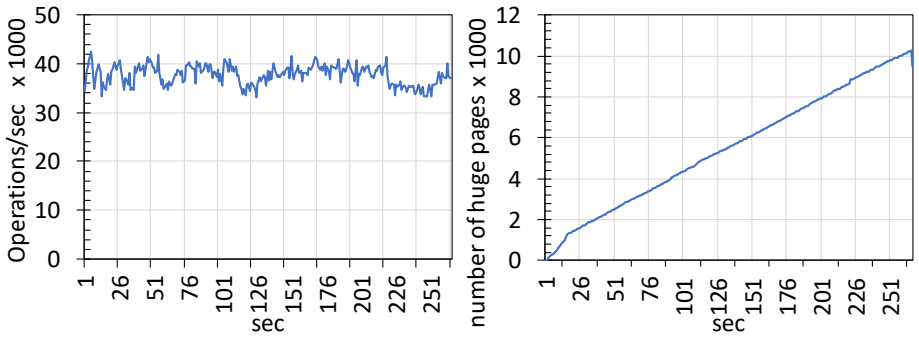
Figure 5.16: 25% Write and 75% Read Performance



(a) Throughput (b) Number of TLB Miss

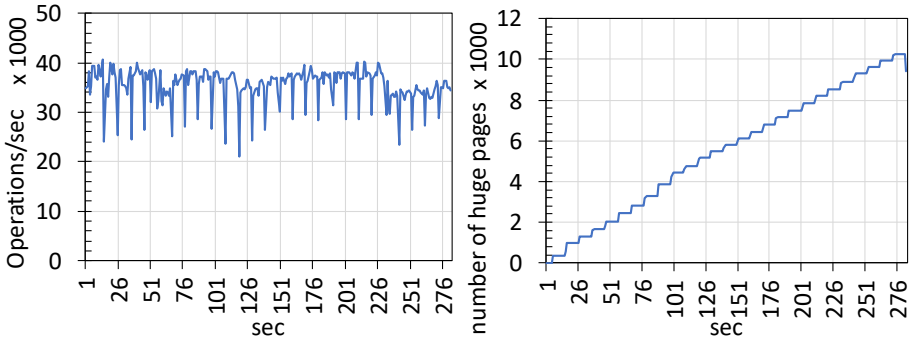
Figure 5.17: 5% Write and 95% Read Performance

to 1.20X over Linux with 4KB base page. Second, using *Xpage* the application throughput is only reduced by 0.63% on average and up to 0.95% compared with Linux with 2MB huge page. Third, using *Xpage* the number of TLB miss is significantly reduced by 24.75% on average and up to 27.25% compared with Linux with 4KB base page. Fourth, using *Xpage* increases the number of TLB miss by 1.01X on average and up to 1.05X over Linux with 2MB huge page. These experiments verify that *Xpage* provides almost the same performance as Linux's 2MB huge page (THP). The performance penalty is very small. The slowdowns stem primarily from accessing OAT, since it is another level of indirection between application and memory allocator. To better understand the overhead of *Xpage*, we measure different types of memory access and compare them, as shown in Table 5.2. *Xpage* memory access latency is reduced by 38.08% compared with Linux 4KB base page. While,



(a) Throughput (b) Number of Huge Pages

Figure 5.18: *Xpage* Performance



(a) Throughput (b) Number of Huge Pages

Figure 5.19: Ingens Performance

Xpage is only 0.46% slower than Linux 2MB huge page.

To compare with the Ingen's utilization-based huge page management approach, we conduct a experiment of loading 10 million key-value pairs generated by YCSB to Redis and measure the real-time Redis throughput and the real-time number of huge pages exist in OS, as shown in Figure 5.18 and Figure 5.19. We highlight two observations. First, the throughput of Redis is flat in *Xpage* case, as shown in Figure 5.18a. But, in Ingens case, the throughput is extremely unstable and deteriorates sharply once asyn promotion occurs, which introduces around 50% instant performance drop, as shown in Figure 5.19a. Second, the number of huge page in *Xpage* case increases smoothly, as shown in Figure 5.18b. In contrast, we observe a step-by-step increase of the number of huge pages in Ingens case, as shown in Figure 5.19b, because of its utilization based promotion design. Each step

Figure 5.20: Number of Huge Page.

corresponds with one performance drop in Figure 5.19a.

5.5.2 Internal Fragmentation

To evaluate *Xpage*'s ability to minimize the internal fragmentation without impacting performance, we evaluate the huge page usage and memory layout, as shown in Figure 5.20, Figure 5.21, and Figure 5.22. To create a parse address space, we first create 100,000 objects, each of them is 1KB and then delete 50% of objects space using a random pattern. Figure 5.20 shows the number of huge pages before and after deleting objects. We observe that both Linux and Ingens keep the same amount of huge page even removing 50% objects, therefore the memory space is 50% fragmented. Only *Xpage* is able to compact memory and release 25 free huge pages back to OS. Specifically, after removing 50% objects, the memory usage of *Xpage* case is reduced to 51.92%, which is very close to the optimal result (50%). *Xpage* only spends 0.013 second to finish the tcache, non-fullrun, and fullrun level memory compaction. The memory space compacted is 100MB. Figure 5.21 shows the memory layout before compaction. Each row is one huge page. The yellow color means occupied region, while blue color means free region. Before compaction, the memory is seriously fragmented, all free memory regions are sparsely distributed across 52 huge pages. After compaction, as shown in Figure 5.22, all objects are packed to low memory address space. The first 25 huge pages are completed freed and could be reclaimed by the OS and used for other processes.

Table 5.3: Memory Utilization of Huge Pages Before Compaction.

Sequence	Occupied (KB)	Unoccupied (KB)
1	105	1934
2 – 50	998	1050
51	982	1066

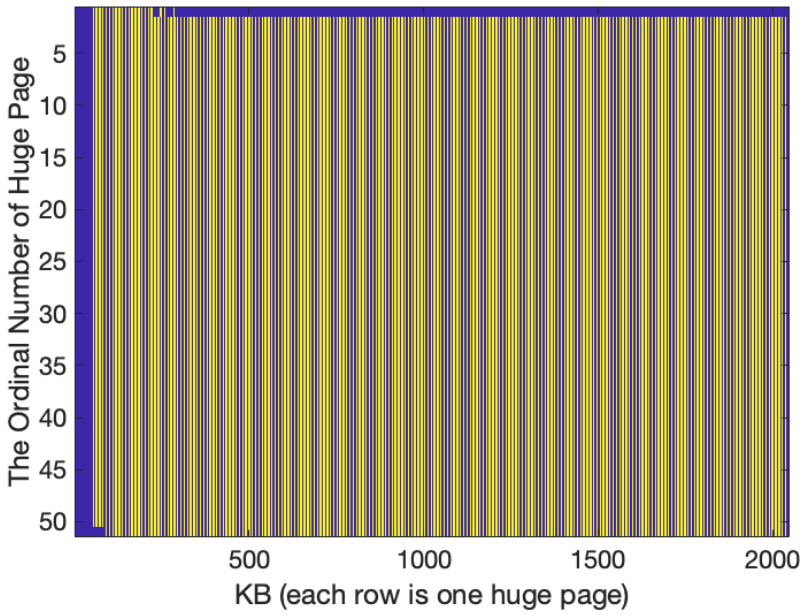


Figure 5.21: Memory Layout before Compaction.

5.6 Related Work

Operating system support for huge pages. Early research discussed the implementation of supporting multiple page size in HP-UX [137]. Navarro et al. [131] implemented support for multiple page sizes in FreeBSD with reservation-based huge page allocations and fragmentation reduction as primary concerns. Carrefour-LP [138] implemented a memory placement algorithm with support for large pages that balance the load across memory controllers and maintain locality. Ingens [133] improves the transparent huge page of Linux by using utilization-based huge page promotion and demotion. Guo et al. [139] improves the huge page deduplication by splitting cold huge pages with high repetition rate and reconstructing split huge pages when they become hot. All previous OS research has focused

Table 5.4: Memory Utilization of Huge Pages After Compaction.

Sequence	Occupied (KB)	Unoccupied (KB)
1 – 25	0	2048
26	121	1927
27 – 50	1996	52
51	1964	84

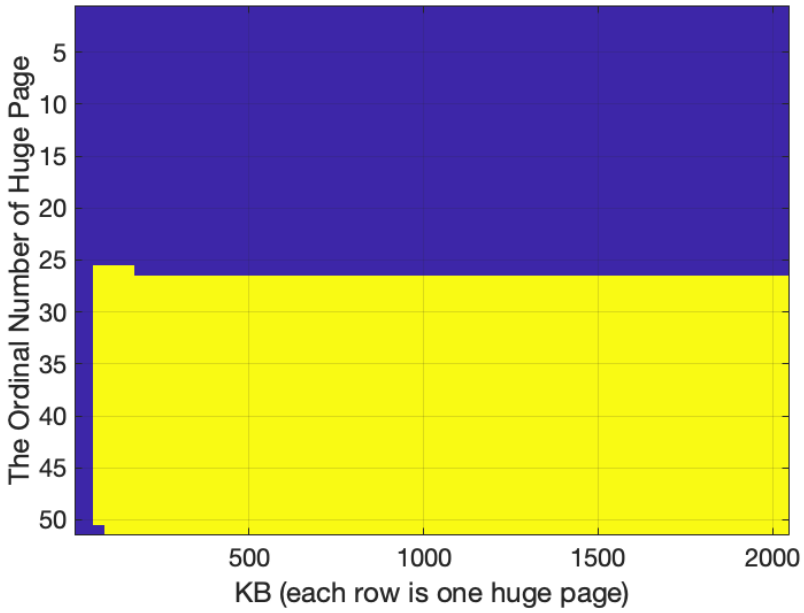


Figure 5.22: Memory Layout after Compaction.

on supporting huge pages, improving memory efficiency, handling external fragmentation. None of them focuses on dealing with internal fragmentation within huge page.

Hardware support for huge pages. Address translation cost can be reduced by improving page walk efficiency [140, 141] or by reducing the number of TLB misses [142, 143, 144, 145, 146, 147, 148, 149, 150, 151]. Mapping multiple pages with a single TLB entry is a way to improve TLB reach by a small factor (max 16X) [152, 146, 147], so this approach still can not deal with the modern memory size. Direct segments [153, 154] map part of a process's linear virtual address space to a contiguous physical memory region, while application modification is required. Most hardware approaches need additional architectural support, but *Xpage* is implemented based on current hardware.

Fragmentation mitigation. Many copying or generational garbage collectors were pre-

sented in user space [155, 156, 157, 158, 159], but these solutions are implemented for Java and not well suited for mitigating huge page internal fragmentation. Optimization in kernel has attracted attention of some researchers. S. Kim et al. [160] presents a proactive anti-fragmentation approach that groups pages with the same lifetime, and stores them in contiguous regions. But this approach only works well for short-lived processes. Current Linux uses memory compaction for defragmenting memory [161], but it leads to performance problems in its current form and is limited to only mitigate external fragmentation.

5.7 Conclusion

We have studied the benefits and adverse effect of huge page in current operating system. To address the problem of performance deterioration and internal fragmentation when huge page are used. We present *Xpage*, a memory management framework that effectively mitigates the fragmentation and makes huge page benefits accessible to application even in stressful conditions. Unlike Linux and Ingens, which simply and aggressively demote huge pages to base pages once fragmentation occurs, *Xpage* never splits huge pages. Instead, it tracks allocated/unallocated memory regions and compacts all memory regions in memory allocator level by placing them carefully within huge page boundaries. *Xpage* is a memory management redesign that brings performance and memory saving to memory intensive applications.

CHAPTER 6

CONCLUSION

Big data and latency-demanding applications [1, 2, 3, 4, 5] are typically deployed using the application deployment models, comprised of virtual machines (VMs), containers, and/or executors/JVMs. These applications enjoy high throughput and low latency if they are served entirely from memory. However, actual estimation and memory allocation are difficult. When these applications cannot fit their working sets in real memory of their VMs/containers/executors, they suffer large performance loss due to excess page faults and thrashing. Even when unused memory is present in other VMs/containers/executors on the same host or a remote node, these applications are unable to share those unused host/remote memory. Moreover, modern servers support terabytes of RAM and memory intensive applications able to take advantage of such large amount of memories are common. But, increased capacity means a significant challenge for address translation.

6.1 Summary

This dissertation makes four unique contributions. First, we have presented a performance evaluation and analysis of in-memory key-value systems. To the best of our knowledge, this is the first in-depth measurement study on critical performance and design properties of in-memory key-value systems, such as the use of different internal data structures, different persistency models and different policies for memory allocators. We measure a number of typical overheads such as memory space, caching, read/write operation performance, fragmentation, and workload throughput performance, to illustrate the impact of different data structures and persistency models on the throughput performance of in-memory key-value systems. We conjecture that the multiple factors on memory efficiency will provide system designers and big data users with a better understanding on how to configure and

tune the in-memory key-value systems for high throughput performances under different workloads and internal data structures.

Second, we have introduced *FastSwap*, a highly efficient shared memory paging facility, with four original contributions. (1) *FastSwap* dynamic shared memory management scheme can effectively utilize the shared memory across VMs through host coordination. (2) *FastSwap* provides efficient support for multi-granularity compression of swap pages in both shared memory and disk swap devices. (3) *FastSwap* provides a hybrid memory swap-out scheme to flush the least recently swap-out pages to disk swap partition when shared memory swap partition reaches a pre-specified threshold and close to full. (4) *FastSwap* provides two level proactive swap-in optimizations (shared memory to memory and disk to shared memory).

In addition, *XMemPod*, a software defined and application transparent disaggregated memory orchestration framework for virtualization of cluster-wide memory, is proposed. *XMemPod* is deployed on a virtualized RDMA cluster without any modifications to user applications and the OSes. It offers orders of magnitude performance gain for bigdata and ML applications. *XMemPod* incurs no functional nor performance impact on guest VMs that do not use it or that incurs zero paging. *XMemPod* provides the ability of dynamically expanding the memory capacity of the virtual machines (VMs) under high memory pressure. By providing a hierarchical memory expansion and sharing framework, *XMemPod* enables memory intensive workloads on a VM to expand its memory demand over virtualized host memory first, and remote memory next, before resorting to external disk. To further improve the utilization and access latency of disaggregated memory, *XMemPod* provides a suite of optimization techniques.

Last but not the least, a huge page management framework, *Xpage*, is designed to mitigate fragmentation and make huge page benefits accessible to applications even in stressful conditions. *Xpage* interacts with both application and kernel to monitor the the degree of huge page fragmentation, which is analyzed to make decision of when to execute tcache,

non-fullurn, and fullrun level compaction. To ensure fairness and security, *Xpage* deals with internal fragmentation within each process's memory space and use each process's CPU time to execute tasks. It is a run-time library and completed isolated from other running process memory space. By avoiding the kernel space overhead and complexity, *Xpage* is the first memory management framework that effectively deals with huge page fragmentation problem in memory allocator level. It is a redesign that brings performance and memory saving to memory intensive applications with dynamic memory behavior.

6.2 Future Work

There are many interesting open research problems for in-memory computing area. Two challenging and promising areas that I am particularly interested to pursue are below.

6.2.1 Disaggregated Memory

Disaggregated memory has attracted much attention recentlly [42, 43, 17, 44, 15, 45]. Most proposals rely on new hardware architecture, new network protocols to cut down the communication cost. Some proposals show the benefit of leveraging RDMA technology [46, 47, 48, 49, 50, 51, 52, 48], exploiting the disk-network latency gap by limiting remote memory to certain workloads, such as remote storage for key-value stores[53, 54, 55, 56], distributed objects [57], swap pages [58, 59, 60, 61, 62, 21], object replication [63]. Most of these efforts lack of desired transparency and all existing proposals treat and leverage unused host memory as the remote memory, fail to take advantage of the performance gap between DRAM and network interconnect, such as Infiniband. Until the network interconnect offers the same level of low latency I/O as the DRAM, we argue that a latency-sensitive disaggregated memory orchestration framework is critical for efficient virtualization of cluster wide memory. I am confident that combining the techniques of disaggregated memory with memory intensive applications is promising in significantly boosting the performance for bigdata and machine learning workload.

6.2.2 Huge Page Management

One open problem of the future huge page management is how to handle internal fragmentation. All previous OS research has focused on supporting huge pages, improving memory efficiency, handling external fragmentation. None of them focuses on dealing with internal fragmentation within huge page. Many copying or generational garbage collectors were presented in user space [155, 156, 157, 158, 159], but these solutions are implemented for Java and not well suited for mitigating huge page internal fragmentation. Optimization in kernel has attracted attention of some researchers. S. Kim et al. [160] presents a proactive anti-fragmentation approach that groups pages with the same lifetime, and stores them in contiguous regions. But this approach only works well for short-lived processes. Current Linux uses memory compaction for defragmenting memory [161], but it leads to performance problems in its current form and is limited to only mitigate external fragmentation.

Another open problem is varied page size. Currently, hardware manufactures support three different paging modes, which are 32-bit, PAE, and 4-level paging. Among them, only 4-level paging mode can be adopted for huge pages, since it supports 4KB, 2MB, and 1GB page size. The limitation of varied page size is on both hardware and operating system paging structure design. If more page size are supported, both operating system and applications can get benefits from that. For example, fragmentation could be significantly reduced, since for each memory allocation OS could always get the best fit page size to store objects. Large object could also be stored in large page for further reducing the address translation latency.

REFERENCES

[1] "Memcached, a distributed memory object caching system," `https://memcached.org`.

[2] "Redis, an in-memory data structure store," `https://redis.io`.

[3] "VoltDB, a translytical in-memory database," `https://github.com/VoltDB/voltdb`.

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[5] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *SoCC*, 2013.

[6] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," in *SIGCOMM*, 2012.

[7] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," in *Communications of the ACM*, 2010.

[8] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. Ryu, and M. Ben-Yehuda, "Applications know best: Performance-driven memory overcommit with ginkgo," in *CloudCom*, 2011.

[9] C. A. Reiss, *Understanding Memory Configurations for In-Memory Analytics*. PhD thesis, University of California, Berkeley, 2016.

[10] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers," in *ACM SIGOPS Operating Systems Review*, 2009.

[11] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *SoCC*, 2012.

[12] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *EuroSys*, 2015.

[13] "HP: The Machine," http://www.labs.hpe.com/research/themachine.

[14] "Intel RSA," http://www.intel.com/content/www/us/en/architecture-and-technology/rsa-demo-x264.html.

[15] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *ISCA*, 2009.

[16] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *HPCA*, 2012.

[17] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation.," in *OSDI*, 2016.

[18] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, "Remote memory in the age of fast networks," in *SoCC*, 2017.

[19] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory.," in *USENIX ATC*, 2015.

[20] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables.," in *OSDI*, 2010.

[21] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap.," in *NSDI*, 2017.

[22] "Accelio based network block device," https://github.com/accelio/NBDX.

[23] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types.," in *NSDI*, 2011.

[24] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in *EuroSys*, 2013.

[25] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *SIGCOMM*, 2014.

[26] Y. Mei, L. Liu, X. Pu, S. Sivathanu, and X. Dong, "Performance analysis of network i/o workloads in virtualized data centers," in *IEEE Transactions on Services Computing*, 2013.

[27] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao, "Who is your neighbor: Net i/o performance interference in virtualized clouds," in *IEEE Transactions on Services Computing*, 2013.

[28] S. Sivathanu, L. Liu, M. Yiduo, and X. Pu, "Storage management in virtualized cloud environment," in *CLOUD*, 2010.

[29] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *SIGCOMM*, 2010.

[30] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing?: Lessons from hundreds of service outages," in *SoCC*, 2016.

[31] C. A. Waldspurger, "Memory resource management in vmware esx server," in *ACM SIGOPS Operating Systems Review*, 2002.

[32] F. Guo, "Understanding memory resource management in vmware vsphere 5.0," Technical Report, VMware, Inc, 2011.

[33] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *ACM SIGOPS Operating Systems Review*, 2004.

[34] W. Zhao, Z. Wang, and Y. Luo, "Dynamic memory balancing for virtual machines," in *ACM SIGOPS Operating Systems Review*, 2009.

[35] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: Monitoring the buffer cache in a virtual machine environment," in *ASPLOS*, 2006.

[36] P. Lu and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache.," in *Usenix ATC*, 2007.

[37] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *Journal of Computer*, 1996.

[38] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *SIGPLAN*, 1990.

[39] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," in *TOCS*, 1989.

[40] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, "Shasta: A low overhead, software-only approach for supporting fine-grain shared memory," in *ACM SIGOPS Operating Systems Review*, 1996.

[41] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain access control for distributed shared memory," in *SIGPLAN*, 1994.

[42] K. Asanovic and D Patterson, "Firebox: A hardware building block for 2020 warehouse-scale computers," in *USENIX FAST*, 2014.

[43] P. Faraboschi, K. Keeton, T. Marsland, and D. S. Milojicic, "Beyond processor-centric operating systems.," in *HotOS*, 2015.

[44] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker, "Network support for resource disaggregation in next-generation datacenters," in *HotNets*, 2013.

[45] P. S. Rao and G. Porter, "Is memory disaggregation feasible?: A case study with spark sql," in *ANCS*, 2016.

[46] F Mietke, R Baumgartl, R Rex, T Mehlan, T Hoefler, and W Rehm, "Analysis of the memory registration process in the mellanox infiniband software stack. 8 2006," in *Euro-Par*, 2006.

[47] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *SIGCOMM*, 2016.

[48] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel® omni-path architecture: Enabling scalable, high performance fabrics," in *HOTI*, 2015.

[49] S.-Y. Tsai and Y. Zhang, "Lite kernel rdma support for datacenter applications," in *SOSP*, 2017.

[50] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," in *ACM SIGCOMM Computer Communication Review*, 2015.

[51] "InfiniBand," `http://www.infinibandta.org`.

[52] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification," Technical Report, 2007.

[53] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *NSDI*, 2014.

[54] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: Distributed transactions with consistency, availability, and performance," in *SOSP*, 2015.

[55] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," in *SIGCOMM*, 2014.

[56] C. Mitchell, Y. Geng, and J. Li, "Using one-sided rdma reads to build a fast, cpu-efficient key-value store.," in *USENIX ATC*, 2013.

[57] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," in *International Workshop on Mobile Object Systems*, 1996.

[58] D. E. Comer and J. Griffioen, "A new design for distributed systems: The remote memory model," Technical Report, Department of Computer Science, Purdue University, 1990.

[59] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, "Implementing global memory management in a workstation cluster," in *ACM SIGOPS Operating Systems Review*, 1995.

[60] M. D. Flouris and E. P. Markatos, "The network ramdisk: Using remote memory on heterogeneous nows," *Journal of Cluster Computing*, 1999.

[61] S. Koussih, A. Acharya, and S. Setia, "Dodo: A user-level system for exploiting idle memory in workstation clusters," in *HPDC*, 1999.

[62] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over infiniband: An approach using a high performance network block device," in *Cluster Computing*, 2005.

[63] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *ACM SIGPLAN Notices*, 2015.

[64] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, N. S. Islam, H. Subramoni, and D. K. Panda, "Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems," in *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, 2012.

[65] O. Paz, "Infiniband essentials every hpc expert must know," in *Retrieved August*, 2014.

[66] "Hbase, an open-source, non-relational, distributed database," `https://hbase.apache.org`.

[67] "MongoDB, a free and open-source cross-platform document-oriented database program," https://www.mongodb.org.

[68] "Cassandra, a free and open-source distributed wide column store NoSQL database management system," http://cassandra.apache.org.

[69] U. Drepper, "What every programmer should know about memory," in *Red Hat, Inc*, 2007.

[70] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," ACM, 1984.

[71] R. B. Hagmann, "A crash recovery scheme for a memory-resident database system," in *IEEE Transactions on Computers*, 1986.

[72] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," USENIX, 2014.

[73] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, *et al.*, "The case for ramclouds: Scalable high-performance storage entirely in dram," in *ACM SIGOPS Operating Systems Review*, 2010.

[74] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the bsdcan conference*, 2006.

[75] "glibc, a GNU C Library project provides the core libraries for GNU/Linux systems," https://www.gnu.org/.

[76] J. Bonwick *et al.*, "The slab allocator: An object-caching kernel memory allocator.," in *USENIX summer*, 1994.

[77] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010.

[78] "Redis Labs. Memtier Benchmark," https://github.com/RedisLabs/memtierbenchmark.

[79] S. Godard, "Sysstat utilities home page," in *pagespersoorange*, 2010.

[80] A. C. De Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, 2010.

[81] R. R. Branco, "Ltrace internals," in *Linux symposium*, 2007.

[82] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, 2007.

[83] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Power and performance evaluation of memcached on the tilepro64 architecture," in *Sustainable Computing: Informatics and Systems*, 2012.

[84] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, 2012.

[85] J. Lindström, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila, "Ibm soliddb: In-memory database optimized for extreme speed and availability.," in *IEEE Data Eng. Bull.*, 2013.

[86] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing.," in *NSDI*, 2013.

[87] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[88] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, *et al.*, "Scaling memcache at facebook.," in *NSDI*, 2013.

[89] "Twemcache: Twitter memcached," `https://github.com/twitter/twemcache`.

[90] R. Birke, L. Y. Chen, and E. Smirni, "Data centers in the wild: A large performance study," Technical Report, IBM Research, 2012.

[91] N. Amit, D. Tsafrir, and A. Schuster, "Vswapper: A memory swapper for virtualized environments," in *SIGPLAN*, 2014.

[92] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "Kvm: The linux virtual machine monitor," in *Proceedings of the Linux symposium*, 2007.

[93] M Kjelso, M Gooch, and S Jones, "Empirical study of memory-data: Characteristics and compressibility," in *IEE Proceedings-Computers and Digital Techniques*, 1998.

[94] R. B. Tremaine, T. B. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy, "Pinnacle: Ibm mxt in a memory controller chip," in *IEEE Micro*, 2001.

[95] M. Oberhumer, “Lzo real-time data compression library,” User Manual for LZO, 2005.

[96] Y Collet, “Lz4–extremely fast compression,” Technical Report, 2015.

[97] Q. Zhang, *Dynamic Shared Memory Architecture, Systems, and Optimizations for High Performance and Secure Virtualized Cloud*. PhD thesis, Georgia Institute of Technology, 2017.

[98] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, “Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark,” in *International Conference on Computing Frontiers*, 2015.

[99] S Jennings, “The zswap compressed swap cache,” in *LWN.net*, 2013.

[100] W. Cao and L. Liu, “Efficient host and remote memory sharing for big data and machine learning applications,” Technical Report, Georgia Institute of Technology, 2018.

[101] Q. Zhang and L. Liu, “Shared memory optimization in virtualized cloud,” in *CLOUD*, 2015.

[102] E. A. Anderson and J. M. Neefe, “An exploration of network ram,” Technical Report, Computer Science Division, University of California, Berkeley, 1998.

[103] H. Chen, Y. Luo, X. Wang, B. Zhang, Y. Sun, and Z. Wang, “A transparent remote paging model for virtual machines,” in *International Workshop on Virtualization Technology*, 2008.

[104] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets, “Cashmere-vlm: Remote memory paging for software distributed shared memory,” in *IPPS/SPDP*, 1999.

[105] E. P. Markatos and G. Dramitinos, “Implementation of a reliable remote memory pager.,” in *USENIX ATC*, 1996.

[106] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel, “Nswap: A network swapping module for linux clusters,” in *European Conference on Parallel Processing*, 2003.

[107] P. Zhang, X. Li, R. Chu, and H. Wang, “Hybridswap: A scalable and synthetic framework for guest swapping on virtualization platform,” in *INFOCOM*, 2015.

[108] M. Hao, G. Soundararajan, D. R. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, “The tail at store: A revelation from millions of hours of disk and ssd deployments.,” in *FAST*, 2016.

[109] K. Elmeleegy, C. Olston, and B. Reed, "Spongefiles: Mitigating data skew in mapreduce using distributed memory," in *SIGMOD*, 2014.

[110] "Emulab," `https://www.emulab.net`.

[111] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," in *HotCloud*, 2010.

[112] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *ICDEW*, 2010.

[113] "Linux memory management," `http://www.tldp.org/LDP/tlk/mm/memory.html`.

[114] A. W. Richa, M Mitzenmacher, and R Sitaraman, "The power of two random choices: A survey of techniques and results," in *Combinatorial Optimization*, 2001.

[115] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems.," in *USENIX ATC*, 2010.

[116] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Journal of Algorithms*, 2004.

[117] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using gpus in software packet processing.," in *NSDI*, 2015.

[118] "Canneal," `http://parsec.cs.princeton.edu/overview.htm`.

[119] "Apache Solr," `http://lucene.apache.org/solr`.

[120] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *SIGMETRICS*, 2012.

[121] P. Sharma and P. Kulkarni, "Singleton: System-wide page deduplication in virtual environments," in *HPDC*, 2012.

[122] D. G. Murray, H Steven, and M. A. Fetterman, "Satori: Enlightened page sharing," in *USENIX ATC*, 2009.

[123] "Frontswap," `https://www.kernel.org/doc/Documentation/vm/frontswap.txt`.

[124] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *MICRO*, 2013.

[125] S. Roy, R. Kumar, and M. Prvulovic, "Improving system performance with compressed memory," in *IPDPS*, 2001.

[126] F. Douglis, "The compression cache: Using on-line compression to extend physical memory.," in *USENIX Winter*, 1993.

[127] Y. Du, M. Zhou, B. Childers, R. Melhem, and D. Mossé, "Delta-compressed caching for overcoming the write bandwidth limitation of hybrid main memory," in *TACO*, 2013.

[128] L. Benini, D. Bruni, A. Macii, and E. Macii, "Memory energy minimization by data compression: Algorithms, architectures and implementation," in *VLSI*, 2004.

[129] N. R. Mahapatra, J. Liu, and K. Sundaresan, "The performance advantage of applying compression to the memory system," in *SIGPLAN*, 2002.

[130] I. C. Tuduce and T. R. Gross, "Adaptive main memory compression," in *USENIX ATC*, 2005.

[131] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," in *ACM SIGOPS Operating Systems Review*, 2002.

[132] "Transparent Hugepages," `https://lwn.net/Articles/359158/`.

[133] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *OSDI*, 2016.

[134] M. Gorman, "Huge pages introduction," in *LWN.net*, 2010.

[135] "libhugetlbfs - Linux man page," `https://linux.die.net/man/7/libhugetlbfs`.

[136] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," 2011.

[137] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath, "Implementation of multiple pagesize support in hp-ux.," in *USENIX Annual Technical Conference*, 1998.

[138] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large pages may be harmful on numa systems," in *USENIX Annual Technical Conference*, 2014.

[139] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. Lui, "Smartmd: A high performance deduplication engine with mixed pages," in *Annual Technical Conference*, 2017.

[140] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," in *ACM SIGARCH Computer Architecture News*, 2010.

[141] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[142] A. Bhattacharjee and M. Martonosi, "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[143] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: An application-driven study," in *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002.

[144] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based tlb preloading," in *International Symposium on Computer Architecture (ISCA)*, 2000.

[145] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly tlb designs," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[146] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[147] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing tlb reach by exploiting clustering in page translations," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[148] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[149] D. Lustig, A. Bhattacharjee, and M. Martonosi, "Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs," in *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[150] S. Srikantaiah and M. Kandemir, "Synergistic tlbs for high performance address translation in chip multiprocessors," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[151] J. Ahn, S. Jin, and J. Huh, "Fast two-level address translation for virtualized systems," in *IEEE Transactions on Computers*, 2015.

[152] M. Talluri and M. D. Hill, "Surpassing the tlb performance of superpages with less operating system support," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[153] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ACM SIGARCH Computer Architecture News*, 2013.

[154] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[155] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker, "Combining generational and conservative garbage collection: Framework and implementations," in *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989.

[156] D. Stefanović, K. S. McKinley, and J. E. B. Moss, "Age-based garbage collection," in *ACM SIGPLAN Notices*, 1999.

[157] G. L. Steele Jr, "Multiprocessing compactifying garbage collection," in *Communications of the ACM*, 1975.

[158] E. Petrank, "A parallel, incremental, mostly concurrent garbage collector for servers," in *ACM Transactions on Programming Languages and Systems*, 2005.

[159] T. Merrifield and H. R. Taheri, "Performance implications of extended page tables on virtualized x86 processors," in *ACM SIGPLAN Notices*, 2016.

[160] S.-H. Kim, S. Kwon, J.-S. Kim, and J. Jeong, "Controlling physical memory fragmentation in mobile systems," in *ACM SIGPLAN Notices*, 2016.

[161] J. Corbet, "Memory compaction," in *LWN.net*, 2010.

VITA

Wenqi Cao was born in Beijing, China. He received his Bachelor degree in electronic and information engineering from Beijing University of Technology, Beijing, China, in July 2007. He received his Master degree in computer science from Pennsylvania State University – University Park, state college, US, in May 2013. From 2007 to 2013, he worked for SIEMENS, HP, and Microsoft respectively. He started his Ph.D. study in College of Computing at Georgia Institute of Technology since August 2015. His research interests are broadly in distributed system and operating system memory management. Specifically, he consistently committed to improve the efficiency of memory management of memory intensive applications, memory allocator, virtualization, Linux kernel, memory disaggregation system, and RDMA network. He has done internships at IBM Almaden Research Center and IBM T.J. Watson Research Center.